UNIVERSITÉ DE MONTRÉAL

CENTRE DE RECHERCHE SUR LES TRANSPORTS

CASE POSTALE 6128, SUCCURSALE A, MONTRÉAL, QUÉBEC, H3C 3J7

CRT-613

LE TRAITEMENT THÉORIQUE ET PRATIQUE DE PARTITIONS POLYGONALES DE L'ESPACE BIDIMENSIONNEL

par

Diane Larin

Mémoire de maîtrise en Informatique, département d'Informatique et de Recherche opérationnelle, dirigé par Michael Florian et Heinz Spiess.

Université de Montréal

Centre de recherche sur les transports - Publication #613

Septembre 1988

Dépôt légal - 1er trimestre 1989

SOMMAIRE

La motivation initiale de ce travail dérive du problème qui consiste à faire des opérations logiques sur des données associées à différentes partitions zonales.

Les frontières de ces zones étant définies par des formes polygonales, nous introduisons d'abord un nouveau concept, soit celui des polygones généralisés. Ces derniers sont constitués par un ensemble de polygones simples dont les frontières ne s'intersectent pas et qui induisent une bipartition des points de l'espace.

Nous justifions l'importance de notre étude en montrant que les polygones généralisés, contrairement aux polygones simples, forment une classe fermée sur les opérations logiques et qu'il n'est pas évident d'effectuer une opération logique entre deux ensembles de polygones, en appliquant simplement les algorithmes qui existent déjà pour deux polygones. Nous analysons les propriétés théoriques de ces polygones généralisés et des trois grandes classes d'opérations s'y rapportant, soit les opérations scalaires, unaires et binaires. Nous proposons ensuite des algorithmes qui calculent directement le résultat d'une opération quelconque, et plus particulièrement d'une opération binaire, entre deux polygones généralisés.

Tous ces concepts théoriques sont à la base du développement d'un outil informatique. Cet outil est composé d'un ensemble de commandes permettant de réaliser les principales opérations sur les polygones généralisés, et d'une bibliothèque de fonctions s'adressant au programmeur intéressé à développer d'autres logiciels traitant le même type d'objets.

Nous présentons cet outil en commençant par la description du format externe et interne permettant de représenter ces ensembles de polygones, puis nous expliquons les détails d'implantation des principales procédures. Nous analysons également divers types de dégénérescence, tout particulièrement le cas des segments parallèles et celui des composantes ne se touchant qu'en un point, et nous proposons une méthode qui permet de traiter de façon satisfaisante la majorité de ces cas dégénérés. Nous décrivons enfin les principes d'utilisation de nos commandes.

TABLE DES MATIERES

SOMMAIRE		iii
LISTE DES TABLEAUX		vi
LISTE DES FIGURES		vii
REMERCIEMENTS		viii
INTRODUCTION		1
I.	LE CONCEPT DE POLYGONE GENERALISE	
I.1	Description du problème	3
I.2	Revue de la littérature	8
I.3	Définitions de base	9
II.	LES ASPECTS THEORIQUES	
II.1	Notation et propriétés de base	12
II.2	Représentation informatique des golygones	16
II.2.1	Arborescence d'inclusion	16
II.2.2	Normalisation	19
II.3	Les opérations scalaires sur les golygones	25
II.3.1	Surface et périmètre	25
II.3.2	Inclusion d'un point	26
II.3.3	Simplicité	28
II.4	Les opérations unaires sur les golygones	30
II.4.1	Complémentarisation	30
II.4.2	Transformations linéaires	30

II.5	Les opérations binaires sur les golygones	35
II.5.1	Propriétés des opérations binaires	35
II.5.2	Algorithmes	38
III.	LES ASPECTS INFORMATIQUES	
III.1	Les structures de données	51
III.1.1	Le format externe	52
III.1.2	Le format interne	54
III.2	Les procédures	60
III.2.1	Entrée/sortie et validation	63
III.2.2	Opérations scalaires	69
III.2.3	Opérations unaires	71
III.2.4	Opérations binaires	74
III.2.5	Problèmes de dégénérescence	80
III.3	Les programmes	87
CONCLUSION		99
REFERENCES		101
APPENDICE A		ix
APPENDICE B		xxi

LISTE DES TABLEAUX

I	Liste des opérations analysées	11
II	Règles du test d'inclusion d'un point dans un golygone	48
III	Liste des fonctions accessibles au programmeur	61
IV	Liste des principales variables utilisées	62
V	Résultats pour diverses approches de traitement de cas dégénérés	85

LISTE DES FIGURES

1.1	Domaine d'appartenance approximé par un polygone	4
1.2	Domaine d'appartenance approximé par plusieurs polygones	5
1.3	Domaine d'appartenance composé	6
1.4	Résultat d'opérations multiples sur des polygones généralisés	7
2.1	Exemple de golygone fini et infini	15
2.2	Golygone avec zones d'appartenance mises en évidence	16
2.3	Trois cas de polygones imbriqués	19
2.4	Trois cas de polygones séparés	20
2.5	Golygone normalisé fini	22
2.6	Arborescence d'inclusion représentant un golygone fini	23
2.7	Ligne de hachurage à travers un golygone fini	28
2.8	Régions d'appartenance de chaque côté d'une ligne frontière	37
2.9	Régions d'appartenance définies par l'intersection de deux segments	38
2.10	Intersection impliquant plus de deux polygones par composante	40
2.11	Union impliquant la relation d'inclusion entre les composantes	41
2.12	Régions d'appartenance induites par diverses combinaisons	45
2.13	Visualisation des zones d'union et d'intersection par hachurage	46
3.1	Organisation des données en mémoire	59
3.2	Cas dégénérés d'inclusion	70
3.3	Divers cas de dégénérescence	81
3.4	Trois combinaisons entre deux segments parallèles	82
3.5	Annotation créée avec le logiciel EMME/2 - Golygone A	95
3.6	Annotation créée avec le logiciel EMME/2 - Golygone B	96
3.7	Résultat de l'union des golygones A et B	97
3.8	Résultat de l'intersection des golygones A et B	98

REMERCIEMENTS

Qu'il me soit d'abord permis d'exprimer ma gratitude envers mes directeurs de recherche, M. Michael Florian, professeur titulaire au département d'informatique et de recherche opérationnelle, et M. Heinz Spiess, chercheur invité au Centre de recherche sur les transports de l'Université de Montréal. Ils m'ont donné toute leur confiance et m'ont prodigué conseils, critiques constructives et encouragements.

Je tiens aussi à remercier tout particulièrement:

Les organismes C.R.S.N.G. et F.C.A.R. pour leur aide financière sans laquelle ce travail n'aurait pas été possible.

Mes collègues, Isabelle Constantin et Denis Lebeuf, pour la lecture attentive qu'ils ont portée à ce texte. Par leurs judicieux commentaires, ils ont grandement contribué à améliorer la version finale de ce mémoire.

Mon amie, Barbara Spiess, pour son accueil toujours chaleureux et ses délicieux repas. Elle a su donner un petit côté de vacances à mes semaines de travail en Suisse.

Mon conjoint, Théodor Crainic. Il est difficile d'exprimer en peu de mots l'importance de l'aide soutenue qu'il m'a apportée, tant au niveau technique qu'au niveau moral.

Mes filles, Lucie et Manon, pour leur amour et leur patience. Malgré mon manque de disponibilité, elles n'ont jamais cessé de me considérer comme la ''meilleure maman du monde''.

Mon père et ma mère à qui je dois tout. Parce qu'ils croyaient sincèrement que le meilleur héritage qu'ils pouvaient me laisser était l'instruction, je voudrais, avec toute ma reconnaissance, leur dédier ce travail.

INTRODUCTION

La plupart des données socio-économiques concernant une région ou un pays sont cueillies et conservées sur une base zonale. Cependant, la définition des zones peut être significativement différente d'un ensemble de données à l'autre. Pensons seulement à la division de la province de Québec en circonscriptions électorales provinciales et fédérales. De plus, chaque ensemble est généralement dédié à un nombre limité d'informations spécifiques.

Il est donc extrêmement utile et important de pouvoir comparer et combiner de tels ensembles et, surtout, d'être en mesure d'extraire l'information pertinente d'un ensemble, par rapport à la définition zonale d'un autre, ou d'une collection d'autres ensembles. Quelques exemples d'applications spécifiques sont les études des caractéristiques de la population à des fins politiques, sociologiques ou de marketing, ainsi que la planification de l'utilisation du territoire et des réseaux de transport.

Conceptuellement, toute définition zonale induit une partition de l'espace considéré. Un type particulier de partition est la partition binaire ou bipartition. Celle-ci consiste à séparer les points de l'espace selon qu'ils satisfassent, ou non, un certain ensemble de critères. Outre son intérêt intrinsèque, l'étude de la représentation géométrique et du traitement des bipartitions est importante puisque celles-ci peuvent aussi servir à traiter des problèmes impliquant d'autres types de partitions.

Les bipartitions de l'espace à deux dimensions peuvent, de façon géométrique, être adéquatement représentées par des ensembles de polygones possiblement imbriqués. Afin de pouvoir les manipuler efficacement, nous voulons pouvoir considérer de tels ensembles comme formant une classe d'objets qui soit fermée sur les opérations logiques, c'est-à-dire que le résultat d'une opération logique sur deux ensembles de polygones donne toujours un objet qui appartienne à la même classe (par opposition à la classe des polygones quelconques qui n'est pas fermée sur les opérations d'union et d'intersection et à la classe des polygones convexes qui ne l'est pas non plus sur l'opération d'union). En effet, l'intersection de deux polygones non

convexes ne donne pas nécessairement un polygone mais peut donner un ensemble de polygones. De même en est-il de l'union de deux polygones, même convexes, lorsqu'ils ne partagent aucun point d'intersection.

L'objet spécifique de ce mémoire est donc de mettre en évidence l'importance de ces ensembles de polygones, possiblement compris les uns dans les autres de façon complexe, d'analyser leurs propriétés théoriques et de montrer comment on peut, de façon informatique, représenter et manipuler ces ensembles. Nous désirons également intégrer toutes ces notions dans le développement d'un outil qui s'adresse à deux niveaux d'utilisateurs, en offrant un ensemble de commandes à l'utilisateur de logiciel qui peut avoir à manipuler des données zonales définies sur une base polygonale, et en offrant une bibliothèque de fonctions au programmeur intéressé à développer d'autres logiciels traitant le même type de données.

Ce mémoire comporte trois chapitres. Dans le chapitre I nous décrivons de façon plus détaillée le problème que nous voulons résoudre, nous en montrons l'originalité en présentant une revue de la littérature et en soulignant la pénurie de références qui s'y rapportent, puis nous introduisons les définitions de base. Dans le chapitre II nous définissons la notation, nous montrons comment représenter de façon mathématique une bipartition et nous étudions les propriétés théoriques de ces partitions. Nous nous intéressons ensuite à trois grandes classes d'opérations sur ces partitions, soit les opérations scalaires, unaires et binaires, et nous présentons les algorithmes servant à réaliser certaines de ces opérations. Le chapitre III contient les détails d'implantation et d'utilisation de l'outil que nous avons développé et qui intègre les notions théoriques présentées au chapitre précédent. Ce chapitre est divisé en trois grandes sections correspondant aux trois niveaux d'interface avec l'outil soit, interface de bas niveau, interface avec le programmeur et interface avec l'utilisateur. Nous y ajoutons également une analyse et une classification des principaux types de dégénérescence et nous suggérons un traitement efficace pour plusieurs d'entre eux. Nous concluons en résumant les principaux apports de ce mémoire et en jetant un regard sur l'avenir. Enfin, nous donnons en appendice le manuel d'utilisation des commandes et des procédures disponibles à ce jour.

I. LE CONCEPT DE POLYGONE GENERALISE

Dans ce chapitre, nous précisons d'abord l'objet de notre étude. Nous présentons également une brève revue de la littérature qui s'y rattache. La présentation des définitions fondamentales utilisées dans ce mémoire complète le chapitre.

I.1 Description du problème

Tout domaine défini par un certain sous-espace de l'espace bidimensionnel peut lui-même être partitionné de plusieurs façons. Pensons par exemple à divers types de cartes, chacune mettant en évidence des caractéristiques différentes des points appartenant à une même région géographique (langue, revenu, densité démographique, altitude, précipitations, type de production agricole, etc...).

Parmi tous les critères de partition possibles, nous nous intéresserons uniquement à ceux induisant une bipartition de l'espace, c'est-à-dire ceux séparant les points qui répondent à un certain critère de sélection, de ceux qui n'y répondent pas. Par exemple, nous pourrions vouloir représenter tous les points de la région étudiée situés à moins de 50 km d'un aéroport et à densité démographique élevée.

Lorsque les sous-domaines ainsi définis forment des régions bornées et continues (i.e. tout point à l'intérieur des limites d'un sous-domaine y appartient), on peut identifier les frontières qui séparent les régions d'appartenance des régions de non-appartenance. Ces frontières peuvent alors être représentées par des figures polygonales.

Le cas le plus simple est celui où les frontières du domaine d'appartenance peuvent être approximées par un seul polygone, comme dans l'exemple illustré à la figure 1.1.

Figure 1.1 Domaine d'appartenance approximé par un polygone

Cependant, un domaine d'appartenance n'est pas nécessairement connexe mais peut être constitué de plusieurs sous-domaines isolés, chacun représenté par sa frontière polygonale, tel qu'illustré à la figure 1.2.

Figure 1.2 Domaine d'appartenance approximé par plusieurs polygones

Dans le cas le plus général, il n'y a pas de limites à la complexité de cette bipartition de l'espace et les domaines composés qui en résultent doivent être représentés par un ensemble de polygones pouvant être, ou non, inclus les uns dans les autres. La figure 1.3 donne un exemple d'un tel domaine composé.

Figure 1.3 Domaine d'appartenance composé

Nous avons donné le nom de **polygones généralisés** à de telles bipartitions.

Au delà de leur représentation, il faut aussi pouvoir manipuler efficacement, aussi bien un seul de ces polygones généralisés, que deux ou plusieurs à la fois. Par exemple, les trois figures précédentes sont toutes définies sur le même espace géographique, englobant la Suisse et les régions limitrophes de France, d'Italie, d'Autriche et d'Allemagne. Ces figures représentent respectivement les frontières de la Suisse (figure 1.1), les principaux lacs (figure 1.2) et les régions à majorité

protestante (figure 1.3). Il faut noter que dans cet espace géographique, toutes les régions sont, officiellement, à majorité soit protestante, soit catholique. Le troisième polygone généralisé représente, donc, en fait une partition entre catholiques et protestants.

Alors, en assumant que les lacs suisses ne sont pas habités, les régions suisses à majorité catholique s'obtiennent par une manipulation appropriée

Suisse *intersection (complément* lacs) *intersection (complément* protestants)

des trois polygones généralisés. Le résultat de cette opération apparaît à la figure 1.4. Ceci n'est qu'un exemple du type de questions qu'on se pose lorsqu'on veut traiter des données socio-économiques à base géographique.

Figure 1.4 Résultat d'opérations multiples sur des polygones généralisés

Le but précis de ce mémoire est donc double. Il s'agit d'abord d'analyser les propriétés théoriques des polygones généralisés et des opérations s'y rapportant, pour ensuite les intégrer dans le cadre du développement d'outils informatiques (Thomas, 1983) qui s'adressent à deux niveaux d'utilisateurs. Dans un premier temps, nous voulons tout d'abord offrir aux utilisateurs, non nécessairement informaticiens, qui cherchent à traiter ce genre de données géo-socio-économiques, un ensemble de commandes. Nous désirons également rendre disponibles les procédures à la base de ces commandes aux programmeurs intéressés à développer d'autres logiciels traitant le même type d'informations.

I.2 Revue de la littérature

Le sujet de ce mémoire, s'intéressant aux polygones et à leur traitement informatique, fait partie de ce qu'on appelle " Computational Geometry". Nous n'avons trouvé aucune référence en français sur ce sujet. Nous proposons donc de traduire le terme " Computational Geometry" par *géométrie numérique*.

Malgré le fait que plusieurs des sujets couverts par la géométrie numérique aient déjà été traités, ce domaine est encore relativement jeune. En font foi, tant un récent livre (Shamos et Preparata, 1985) qui présente une approche unifiée des résultats des travaux dans le domaine, surtout ceux de la dernière décade, qu'un recueil d'articles (Toussaint, 1985) sur les développements les plus récents en géométrie numérique.

Un des sujets largement traité dans cette discipline concerne les opérations sur un ou deux polygones (Bentley et Ottman, 1979; Hohenbalken, 1985; Knott, 1987; O'Rourke et coll., 1982; Shamos et Preparata, 1985; Shamos et Hoey 1976). Lorsqu'il s'agit de faire l'intersection de deux polygones, les différences entre ces auteurs se situent surtout au niveau de l'implantation et non pas au niveau des concepts de base. Nous avons d'ailleurs adapté plusieurs de ces mêmes concepts (voir chapitre II) dans ce travail. Toutefois, en dépit de nos efforts (Toussaint, 1987), nous n'avons trouvé aucune référence traitant directement du sujet auquel nous nous sommes intéressés, soit les polygones généralisés, leurs propriétés et les opérations ayant ces polygones généralisés comme opérandes.

Parmi les auteurs mentionnés, seul Hohenbalken (1985) remarque que le résultat d'une opération logique sur deux polygones peut être constitué de plusieurs polygones. Sans donner plus de détails et sans le traiter, il affirme que ce fait ne pose toutefois pas de problèmes. Cependant, comme nous le montrerons dans le chapitre II, on ne peut pas réaliser une opération logique sur deux polygones généralisés en les considérant simplement deux à deux, dans n'importe quel ordre. En effet, pour qu'on puisse traiter ce cas en considérant les polygones deux à la fois, il faut avoir de l'information sur les relations qui existent entre tous les polygones composant ces polygones généralisés, information qui doit être obtenue par un algorithme spécifique.

La seule autre référence qui mentionne des ensembles de polygones (Knott, 1987) suggère la possibilité d'utiliser un algorithme de remplissage pour faire l'intersection de deux ensembles de polygones qui se chevauchent. Cette approche n'est, ni très pratique, ni généralisable. Toutefois, le fait qu'on la suggère illustre encore une fois la rareté des travaux dans le domaine.

Mentionnons aussi, même si cette approche ne fait pas partie de la géométrie numérique, que certains aspects du problème (les opérations logiques) peuvent être traités par manipulation directe des ''bit maps'' (SPANS, 1988). Outre que cette approche exige une grande quantité de mémoire, elle n'est pas non plus facilement généralisable.

I.3 Définitions de base

Formellement, nous définissons un polygone généralisé comme un sous-ensemble des points de l'espace bidimensionnel défini par un ensemble de polygones simples et dont les frontières ne s'intersectent pas. Pour alléger la nomenclature utilisée dans ce mémoire, nous utiliserons dorénavant l'abbréviation **golygone** pour désigner un polygone généralisé.

Les polygones constituant un golygone peuvent être de deux types: ceux qui délimitent vers l'intérieur, que nous appellerons polygones **positifs**, et ceux qui délimitent vers l'extérieur, que nous appellerons polygones **négatifs**.

Les lignes de contour des polygones forment l'ensemble des points frontière qui séparent les régions ouvertes† d'appartenance et, par conséquent, deux points qu'on peut relier par un chemin qui n'intersecte aucune frontière ont toujours le même état d'appartenance. Un golygone est **réalisable** si, pour toute paire de points de l'espace séparés par exactement une frontière, l'état d'appartenance de l'un diffère de celui de l'autre. Ainsi, l'état d'appartenance de tous les points non frontière est déterminé par la connaissance de l'état d'un seul de ces points et, pour un golygone fini, on peut toujours prendre comme point de référence un point tendant vers l'infini qui, par définition, sera toujours en état de non-appartenance. Enfin, puisque les points de frontière sont connus, la question de leur état d'appartenance n'a pas à être considérée.

Outre sa définition formelle en termes d'ensembles, un golygone est aussi un objet qu'on peut mesurer, transformer, tester. De plus, tel qu'exemplifié au début de ce chapitre, on veut pouvoir combiner plusieurs golygones par des opérations logiques. Nous avons divisé les opérations intéressantes en trois grandes classes: les opérations scalaires, unaires et binaires. La liste de toutes les opérations auxquelles nous nous sommes intéressés dans le cadre de ce travail est donnée au tableau I.

En adaptant sa définition classique à notre contexte, une **opération scalaire** reçoit un golygone en entrée et donne une valeur scalaire en sortie. Des exemples de telles opérations sont: calculer le périmètre ou la surface d'un golygone, trouver son centre de gravité, vérifier la non-intersection de ses frontières, tester l'inclusion d'un point, etc.

Les **opérations unaires** reçoivent un golygone en entrée et donnent un golygone en sortie. Des exemples de telles opérations sont la complémentarisation, qui change l'orientation de tous les segments du golygone, et les transformations linéaires, qui changent la géométrie du golygone d'entrée en appliquant une même suite de modifications à tous ses sommets.

---

† Le terme *ouvert* est employé ici dans le sens mathématique.

Lorsque les opérations sont séparables sommet par sommet, il n'est pas plus difficile de les appliquer aux golygones qu'aux polygones. Par contre, la réalisation d'opérations binaires entre deux golygones présente un plus grand défi.

Les **opérations binaires** sont, en fait, des opérations logiques. Le résultat d'une opération logique sur deux golygones est obtenu en appliquant, point par point, pour tout point de l'espace bidimensionnel, une fonction logique à deux variables binaires. Le résultat d'une opération logique, telle l'intersection ou l'union, n'est pas fermée sur la classe des polygones lorsque ceux-ci ne sont pas convexes. En effet, il est alors possible d'obtenir non pas un polygone, mais un ensemble de polygones. Par contre, nous verrons que les golygones forment une classe d'objets fermée sous les opérations binaires car, en appliquant une de ces opérations à deux éléments de cette classe, on obtient un troisième élément appartenant à la même classe. De plus, les deux opérations binaires d'intersection et d'union, combinées à l'opération unaire de complémentarisation, forment un système d'opérations logiques complet permettant de réaliser n'importe quelle fonction logique.

Opérations scalaires	calcul du nombre de sommets calcul du nombre de composantes calcul du périmètre calcul de la surface test de simplicité test d'inclusion d'un point
Opérations unaires	normalisation complémentarisation transformations linéaires
Opérations binaires	intersection union

Tableau I Liste des opérations analysées

II. LES ASPECTS THEORIQUES

Dans ce chapitre, nous introduisons d'abord la notation avant de traiter des aspects théoriques qui ont guidé l'élaboration des principaux algorithmes. Nous examinons également la justification de ces algorithmes et nous analysons leur complexité.

II.1 Notation† et propriétés de base

Soit $\mathbf{R}^2$ l'espace des réels à deux dimensions et soit a un segment de droite. La queue de ce segment sera notée $\dot{a}$ et la tête $\hat{a}$. Evidemment, $\dot{a}$ et $\hat{a}$ appartiennent à $\mathbf{R}^2$ et sont définies par des coordonnées (x,y). Soit maintenant un deuxième segment $b = (\dot{b}, \hat{b})$. Nous définissons $\Psi(a,b)$ comme le point où la droite supportant le segment a intersecte avec celle supportant le segment b. En exprimant les droites sous leur forme paramétrique, nous avons que:

$$\Psi(a,b) = \dot{a} + \lambda_a(b)\,(\hat{a} - \dot{a}) = \dot{b} + \lambda_b(a)\,(\hat{b} - \dot{b})$$

où $\lambda_a(b)$ et $\lambda_b(a)$ sont les paramètres qui définissent l'intersection. Par convention, lorsque a et b sont parallèles, nous imposerons $\lambda_a(b) = \lambda_b(a) = \infty$, et cela même si a et b se chevauchent. Nous expliquerons la raison de cette convention dans le chapitre traitant de l'implantation des procédures.

Puisque l'intersection des droites n'implique pas nécessairement celle des segments, définissons alors la fonction $\delta_{(a,b)}$ qui signale l'intersection au niveau des segments:

$$\delta_{(a,b)} = \begin{cases} 1 & \text{si } 0 \le \lambda_a(b) \le 1 \quad \text{et} \quad 0 \le \lambda_b(a) \le 1 \\ 0 & \text{autrement} \end{cases}$$

Ainsi, les segments a et b intersectent lorsque $\delta_{(a,b)} = 1$.

† Dans cette section, toute expression nouvelle ou employée dans un sens non classique est signalée par le symbole *.

Considérons maintenant le cas des polygones. Un polygone p est une partition de l'espace définie comme suit:

$$p = \{x \in \mathbf{R}^2 \mid x \text{ appartient au sous-espace délimité par la suite d'arêtes } a_0, a_1, \cdots, a_{m-1}, \text{ où } a_0 = (v_0, v_1),\ a_1 = (v_1, v_2),\ \cdots,\ a_{m-1} = (v_{m-1}, v_1)\}.$$

En termes plus concis, nous nous référerons au polygone p par la suite ordonnée de ses sommets:

$$p = (v_0, v_1, \cdots, v_{m-1}),\quad v \in \mathbf{R}^2 \tag{1}$$

Les points sur les segments forment la frontière du polygone et la frontière délimite les différentes régions d'appartenance. Par rapport à la direction "queue → tête" d'un segment, la région d'appartenance est à gauche et celle de non-appartenance à droite.

Le polygone p est **simple** lorsque $\delta_{(a,b)} = 0$, pour toute paire de segments a, b, non consécutifs dans p. $\mathbf{P}$ représente l'ensemble des polygones simples. Pour fin de concision, lorsque, dans les pages qui suivent, nous emploierons le terme polygone, il sera toujours sous-entendu que nous parlons de polygone simple, à moins de spécification contraire.

Il y a deux types de polygones simples: les **positifs** et les **négatifs**. Par convention, un polygone est dit positif si ses sommets sont ordonnés en sens antihoraire. Soit $|p|$ représentant l'intérieur géométrique du polygone obtenu en faisant abstraction de l'orientation des sommets de p. Nous dirons qu'un point $r \in \mathbf{R}^2$ est **naïvement inclus*** dans p si $r \in |p|$. Définissons maintenant un polygone positif (ou négatif), non pas en fonction de l'ordonnancement de ses sommets, mais en fonction des points de l'espace qu'il inclut. p est positif s'il inclut tous les points contenus dans $|p|$ et n'en inclut aucun autre. On dira alors que p délimite vers l'intérieur. La fonction d'inclusion d'un point $r \in \mathbf{R}^2$, dans un polygone p est notée $\gamma(r, p)$. Pour p positif:

$$\gamma(r, p) = \begin{cases} 1 & \text{si } r \in p \quad (r \in |p|) \\ 0 & \text{si } r \notin p \quad (r \notin |p|) \end{cases} \tag{2}$$

De façon symétrique, p est négatif si ses sommets sont orientés en sens horaire. Alors, p exclut les points qui sont dans $|p|$ et inclut les autres; il délimite donc sa région d'appartenance vers l'extérieur. La fonction $\gamma(r, p)$ est, pour un p négatif:

$$\gamma(r, p) = \begin{cases} -1 & \text{si } r \notin p \quad (r \in |p|) \\ 0 & \text{si } r \in p \quad (r \notin |p|) \end{cases} \tag{3}$$

Les polygones qui ne satisfont aucune de ces deux conditions ($|\gamma(r, p)| > 1$) ne sont pas simples.

Il nous reste à définir les relations qui peuvent exister entre un polygone p et un polygone q.

- p **intersecte** q s'il existe au moins un segment $a \in p$ et un segment $b \in q$ tel que $\delta_{(a,b)} = 1$; sinon p et q sont **f-disjoints***. Il faut remarquer que deux polygones f-disjoints ne sont pas nécessairement disjoints au sens des ensembles (Stoll, 1963) de points inclus par chacun d'eux, mais l'ensemble des segments formant les frontières de l'un est disjoint de l'ensemble des segments formant les frontières de l'autre.
- p et q sont **séparés*** si p et q sont f-disjoints et si $|p| \not\subset |q|$ et $|q| \not\subset |p|$.
- p est **imbriqué*** dans q si p et q sont f-disjoints et si $|p| \subset |q|$.

Soit maintenant $\mathbf{G}$ l'ensemble des golygones. Un golygone $g \in \mathbf{G}$ est défini comme un ensemble non ordonné de composantes polygonales:

$$g = \{ p_0, p_1, \cdots, p_{n-1} \}, \quad p_i \in \mathbf{P}. \tag{4}$$

Il en suit que $g \subseteq \mathbf{R}^2$ et $\mathbf{P} \subset \mathbf{G}$. Nous pouvons aussi partitionner g en deux sous-ensembles disjoints: $g = \{ g^+, g^- \}$, où g^+ est l'ensemble des polygones positifs et g^-, l'ensemble des polygones négatifs.

Comme pour les polygones, les frontières des golygones délimitent les différentes régions d'appartenance.

Un golygone g est **fini** si, pour tout $p \in g^-$, il existe un $q \in g^+$, $p, q \in \mathbf{P}$ tel que $r \in |p| \Rightarrow r \in |q|$, $\forall\, r \in \mathbf{R}^2$. Tout point situé à l'infini est donc exclu d'un golygone fini. Inversement, lorsqu'un point situé à l'infini est inclus dans un golygone, ce dernier est dit **infini**.

La figure 2.1 donne un exemple de golygone fini (A) et infini (B).

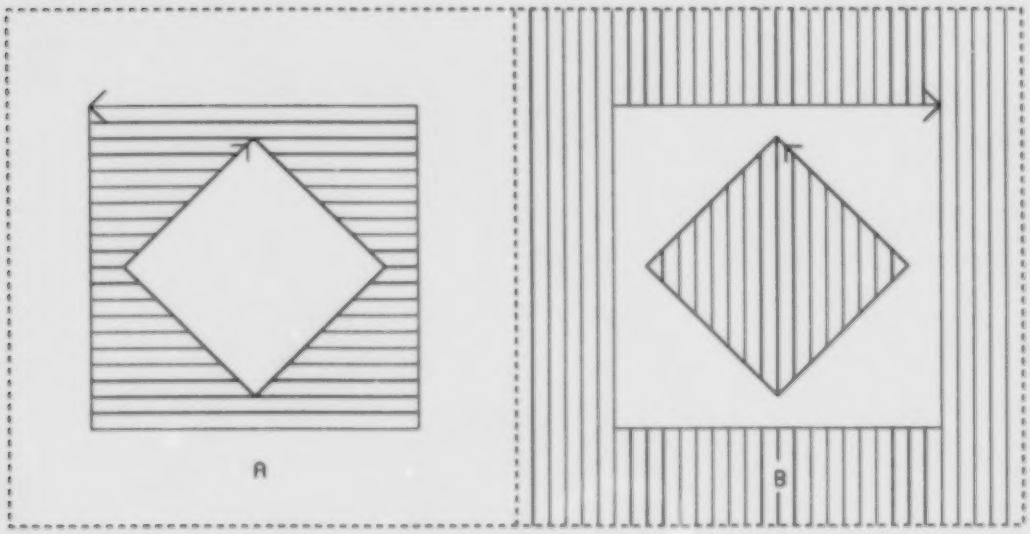


Figure 2.1 Exemple de golygone fini et infini

Définissons enfin les relations qui peuvent exister entre polygones et golygones. Soit $p, q, t \in \mathbf{P}$ et $g \in \mathbf{G}$.

- p **intersecte** g s'il existe au moins un $q \in g$ tel que p intersecte q; sinon p est **f-disjoint*** de g.
- p est **séparé*** de g si, $\forall\ q \in g$, $|p| \not\subset |q|$ et $|q| \not\subset |p|$.
- p est **imbriqué*** dans g s'il existe un $q \in g$ tel que $|p| \subset |q|$.
- p **englobe immédiatement*** $q \in g$ si q est imbriqué dans p et s'il n'existe aucun $t \in g$ tel que $|q| \subset |t| \subset |p|$.

II.2 Représentation informatique des golygones

II.2.1 Arborescence d'inclusion

Dans le chapitre I, nous avons défini un golygone comme étant essentiellement un ensemble de polygones f-disjoints pouvant être, ou non, englobés les uns dans les autres. La figure 2.2 donne un exemple de golygone, les parties hachurées représentant les zones d'appartenance.

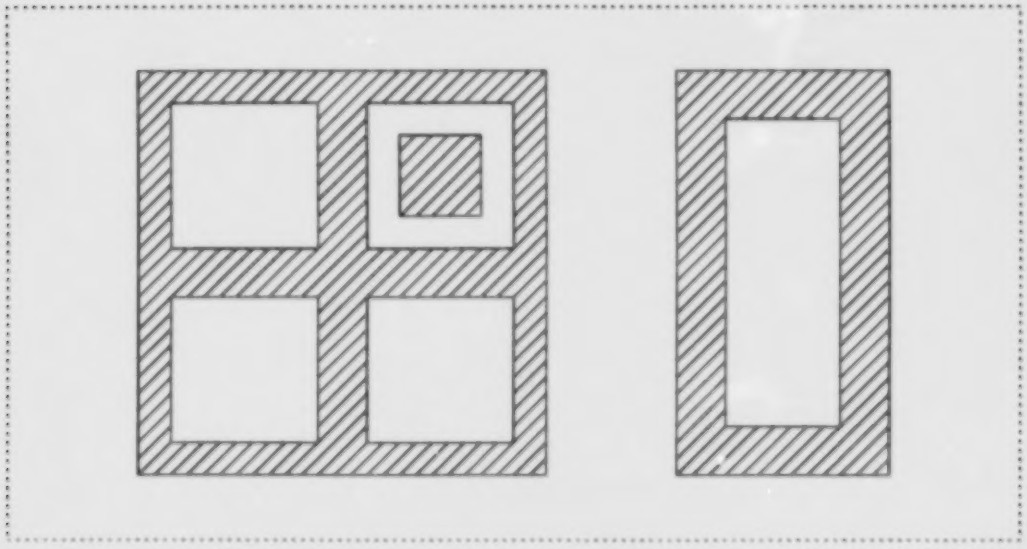


Figure 2.2 Golygone avec zones d'appartenance mises en évidence

Un tel objet forme une structure d'ensembles et de sous-ensembles qui peut facilement être décrite par une **forêt**, c'est-à-dire une collection d'arbres. Parmi les nombreuses façons de représenter une telle structure, l'arbre binaire obtenu en reliant tous les fils d'une même famille et chaque père à son premier enfant (Knuth, 1973) est celle qui apparaît la plus naturelle. L'algorithme 1 sert à bâtir un tel arbre binaire. Il s'agit d'insérer un à un, dans une arborescence au départ vide, tous les éléments d'un ensemble non-ordonné, de façon à ce que tous les ancêtres de l'élément inséré soient des éléments plus "grands", et tous ses descendants, des éléments plus "petits" que lui. Après avoir trouvé sa place dans l'arbre, chaque

élément pointe, à gauche, vers son premier enfant (aîné) et à droite, vers son premier frère (cadet). Remarquons que cet algorithme n'est pas spécifique au contexte des golygones, mais s'applique à tout ensemble dont on peut connaître les relations de descendance entre les éléments. Dans notre cas, la relation de descendance en est une d'inclusion naïve. C'est pourquoi, dans les pages qui suivent, nous parlerons d'**arborescence d'inclusion**.

Algorithme 1: Bâtir l'arborescence d'inclusion d'un golygone g à n polygones p.

ETAPE 0: [Initialiser l'arborescence T]
T := *racine bidon*

ETAPE 1: [Tamiser chacun des éléments dans l'arborescence T]
pour $p = p_1, p_2, \cdots, p_n$ faire
inclure p dans T en appliquant les étapes 2 à 5

ETAPE 2: q := *racine*(T); q' := *aîné*(q)

ETAPE 3: [Trouver la place de p dans T]
tantque ($\exists$ q') faire
si (p est descendant de q') alors
$q := q'$; q' := *aîné*(q)
sinon
q' := *cadet*(q')

ETAPE 4: [Tester si p est descendant ou frère du dernier parent visité]
si (p est descendant de q) alors
q' := *aîné*(q); *aîné*(q) := p
sinon
cadet(p) := *cadet*(q); *cadet*(q) := p
ARRETER

ETAPE 5: [Tester l'inclusion des enfants de q dans p]
tantque ($\exists$ q') faire
tmp := *cadet*(q')
si (q' est descendant de p) alors
cadet(q') := *aîné*(p); *aîné*(p) := q'
sinon
cadet(q') := *cadet*(p); *cadet*(p) := q'
q' := *tmp*

Soit n, le nombre de polygones et m, le nombre total de sommets dans le golygone. Alors, dans le pire cas, on peut bâtir cette arborescence dans $O(mn)$. En

effet, on sait que tester l'inclusion d'un point dans un polygone se fait dans l'ordre du nombre de segments dans ce polygone. Tester l'inclusion d'un polygone dans l'arborescence courante se fait, en pire cas, en testant l'inclusion d'un point quelconque du polygone dans chacun des polygones insérés jusque là, donc $O(m)$. Enfin, puisque cet algorithme est répété pour les n polygones, nous obtenons au total une procédure dans l'ordre de $O(n) \times O(m)$ soit $O(mn)$.

Il y a cependant moyen d'optimiser la recherche de l'arborescence d'inclusion par une implantation plus efficace du test de descendance. Par exemple, il est possible de trier les composantes selon la valeur absolue de leur surface. Ce calcul se fait dans $O(m)$ et un bon tri dans $O(n \ log(n))$, ce qui est négligeable puisque n est beaucoup plus petit que m. Ensuite, si on insère les éléments dans l'arborescence en partant du plus petit jusqu'au plus grand, les étapes 3 et 4 ne sont plus nécessaires puisqu'un élément à insérer ne peut jamais être le descendant d'un des éléments déjà inclus dans l'arborescence. En moyenne, ce traitement préalable peut entraîner une réduction importante du temps de calcul.

II.2.2 Normalisation

Tout ensemble de polygones simples et f-disjoints ne forme pas nécessairement un golygone réalisable. Cependant, comme nous le verrons dans cette section, on peut toujours le rendre réalisable en réorientant certaines de ses composantes. C'est le processus de la **normalisation**.

Un golygone est réalisable si, pour tout point de l'espace, il est possible de définir de façon unique l'état d'appartenance de ce point en fonction de son inclusion dans les polygones avoisinants. Pour des fins d'illustration, considérons un golygone g composé de deux polygones p et q et notons R_{pq} la région comprise entre les frontières de p et de q.

Supposons d'abord p et q positifs, p imbriqué dans q (figure 2.3-A) et considérons un point r situé dans la région R_{pq}. Alors, puisque r est inclus dans q, r devrait appartenir à R_{pq}. D'un autre côté, r n'est pas inclus dans p et par conséquent ne peut pas appartenir à R_{pq}. Il y a donc contradiction et il est impossible de décider si r appartient ou non à R_{pq}. Puisque $R_{pq} \subseteq g$, on ne peut pas non plus décider de l'état d'appartenance de r par rapport à g.

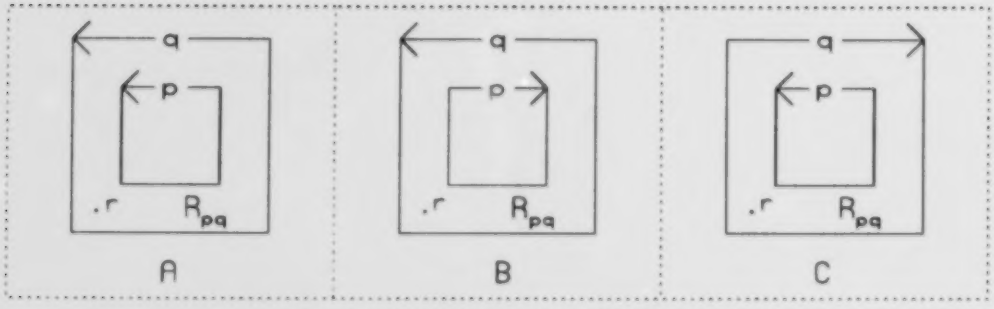


Figure 2.3 Trois cas de polygones imbriqués

Par contre, si p et q sont de signes différents, alors seulement deux cas sont possibles:

a) soit r est inclus dans p et dans q (figure 2.3-B);

b) soit r n'est inclus ni dans p, ni dans q (figure 2.3-C).

Dans les deux cas, il n'y a aucun conflit et la connaissance de l'état d'appartenance du point par rapport à chacun des polygones délimitant la région dans laquelle il est géométriquement situé, définit de façon unique s'il est inclus dans R_{pq} et donc dans g :

- le cas a implique r appartient à R_{pq} et g ;
- le cas b implique r n'appartient ni à R_{pq}, ni à g.

Supposons maintenant p positif, q négatif, p et q séparés, et considérons deux points $r' \in |p|$ et $r'' \in |q|$ (figure 2.4-A). Puisque r' est inclus dans p, il appartient à la région définie par $|p|$. Cette région étant un sous-ensemble de g, r' devrait donc également appartenir à g. En appliquant le même raisonnement, r'' n'étant pas inclus dans q, il ne devrait pas non plus appartenir à g. Que peut-on dire d'un troisième point r situé dans la région R_{pq} séparant p et q? S'il appartient à R_{pq} alors il a, par rapport à g, le même état d'appartenance que r', et s'il n'appartient pas à R_{pq}, il a le même état d'appartenance que r'', ce qui entre en contradiction avec la définition d'un golygone réalisable, puisque $R_{pq} \subseteq g$.

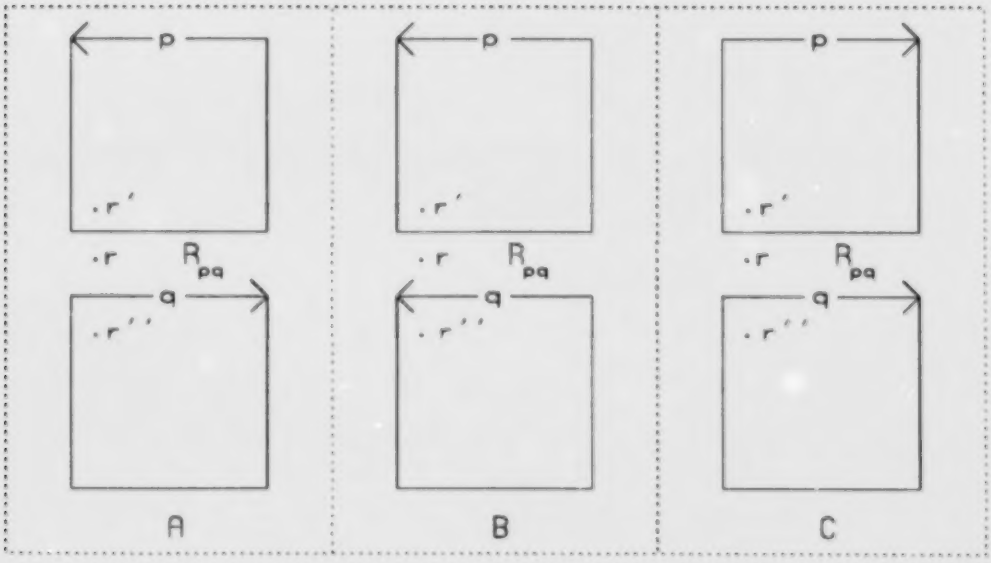


Figure 2.4 Trois cas de polygones séparés

Par contre, si p et q sont de même signe, les seuls cas possibles sont:

a) soit r' appartient à p et r'' appartient à q (figure 2.4-B);

b) soit r' n'appartient pas à p et r'' n'appartient pas à q (figure 2.4-C);

Dans les deux cas, il n'y a plus de conflit puisque:

- le cas a implique r n'appartient ni à p, ni à q, ni à R_{pq};
- le cas b implique r appartient à p, à q et à R_{pq}.

Dans les deux exemples précédents nous ne considérions que des golygones à deux composantes, et nous avons vu que le signe de ces composantes pouvait conduire à des contradictions relatives à l'état d'appartenance de certains points par rapport à ces composantes ainsi qu'au golygone formé par celles-ci. Il est évident que pour un golygone plus complexe, les mêmes problèmes peuvent se présenter. Afin qu'un golygone permette de refléter correctement, et l'état d'appartenance des points aux diverses régions délimitées par les frontières des polygones, et les variations de ces états d'appartenance entre deux régions voisines, nous avons imposé les règles suivantes:

1) tous les polygones qui ne sont imbriqués dans aucun autre doivent avoir le même signe;

2) tous les polygones imbriqués dans un autre doivent avoir un signe différent du polygone immédiatement englobant.

Ces règles assurent que tous les points d'une région délimitée par la frontière de deux polygones, soit appartiendront à ces deux polygones et au golygone, soit n'appartiendront ni au golygone, ni à aucun des deux polygones. Le même raisonnement s'applique pour les points d'une région délimitée par un seul polygone.

L'action d'adapter un golygone à ces conventions s'appelle **normalisation**. Un golygone peut être normalisé fini ou infini selon l'orientation choisie pour les polygones qui ne sont imbriqués dans aucun autre. La figure 2.5 donne un exemple de golygone normalisé fini.

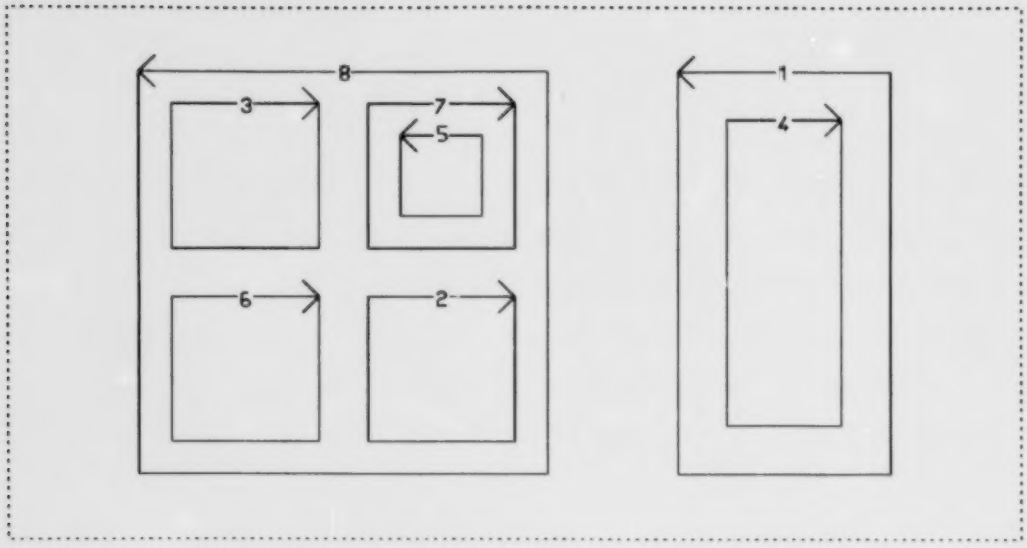


Figure 2.5 Golygone normalisé fini

L'algorithme 2 décrit la procédure à suivre pour obtenir un golygone normalisé fini. La première phase consiste à construire l'arborescence d'inclusion des n polygones formant g, alors que la seconde détermine l'orientation adéquate de chaque polygone en fonction de son niveau dans cette arborescence.

Algorithme 2: Normalisation d'un golygone g fini composé de n polygones p

ETAPE 0: [Initialisation]
$\{ g^+ \} := \emptyset$; $\{ g^- \} := \emptyset$

ETAPE 1: [Construire l'arborescence d'inclusion]
appliquer l'algorithme 1

ETAPE 2: [Orienter correctement chacun des $p \in g$]
pour $j = 1, 2, \cdots, n$ faire
si p_j est situé à un niveau pair alors
$\{ g^- \} := \{ g^- \} \cup p_j$
sinon
$\{ g^+ \} := \{ g^+ \} \cup p_j$

Suite à la normalisation, on peut donc représenter le golygone comme une arborescence binaire dont chaque noeud représente une composante. La racine

correspond à l'espace bidimensionnel. Le lien entre une composante p de niveau m et ses enfants q_p de niveau $m + 1$ représente l'espace

$$|p| \cap \overline{\{\bigcup |q_p|\}}$$

c'est-à-dire l'ensemble des points qui sont inclus dans $|p|$ mais naïvement exclus de tous les enfants q_p de p. Tous les points représentés par l'ensemble des liens reliant les composantes de deux niveaux successifs ont le même état d'appartenance, soit 1 lorsque les liens relient des composantes positives à des composantes négatives, et 0 dans le cas contraire. La figure 2.6 donne l'arborescence d'inclusion correspondant au golygone de la figure 2.5.

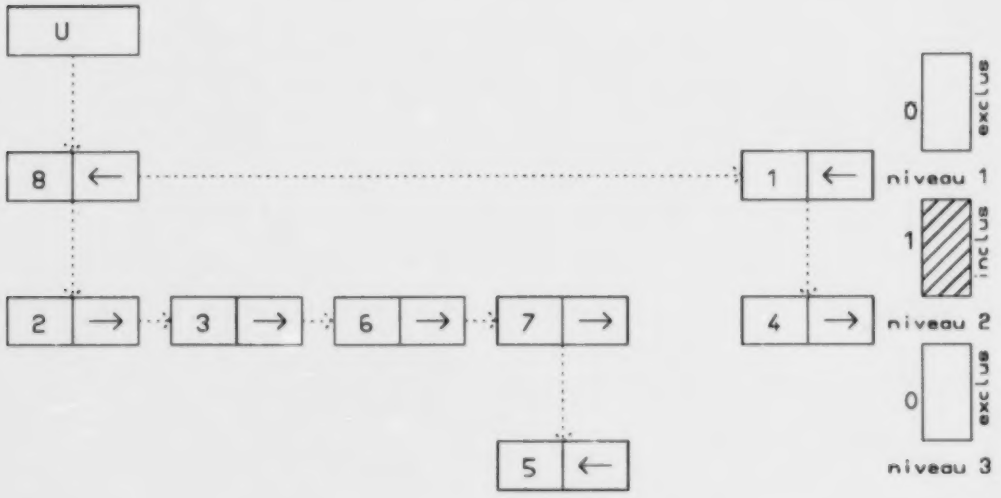


Figure 2.6 Arborescence d'inclusion représentant un golygone fini

Il est donc possible d'exprimer tout golygone simple et normalisé en termes d'union, d'intersection et de complémentarisation des polygones qui le constituent. Il y a une correspondance directe entre les niveaux de parenthèses et ceux de

l'arborescence. De plus, chaque polygone complémenté indique un polygone négatif ce qui, pour notre exemple, donne:

$$\left[|p_8| \cap \overline{(|p_2| \cup |p_3| \cup |p_6| \cup (|p_7| \cap \overline{(|p_5|)}))} \right] \cup \left[|p_1| \cap \overline{(|p_4|)} \right]$$

En utilisant la loi de deMorgan pour simplifier, nous obtenons l'**expression booléenne caractéristique** du golygone, décrivant les zones d'appartenance de ce golygone.

$$\left[p_8 \cap p_2 \cap p_3 \cap p_6 \cap (p_7 \cup p_5) \right] \cup (p_1 \cap p_4)$$

Notons que l'expression booléenne caractéristique d'un golygone normalisé ne peut être obtenue qu'en analysant la structure arborescente de ce golygone.

Dans la suite de ce chapitre on supposera toujours que, à moins de spécification contraire, les golygones sont simples et normalisés.

II.3 Les opérations scalaires sur les golygones

II.3.1 Surface et périmètre

La surface d'un polygone est calculée, comme le propose Von Hohenbalken (1985), par une implantation discrète du théorème de Green sur les intégrales de lignes.

En considérant la suite ordonnée des vecteurs sommets, il s'agit de calculer les déterminants de toutes les matrices de dimension deux obtenues en prenant, deux à deux, chaque paire de sommets consécutifs. L'aire est égale à la moitié de la somme de ces déterminants. Soit x_i et y_i les coordonnées du sommet v_i, $i = 0, ..., m$, avec $v_m = v_0$. La formule servant à calculer la surface du polygone p contenant m sommets est alors:

$$\Lambda (p) = \tfrac{1}{2} \sum_{i=0}^{m-1} (x_i \, y_{i+1} - x_{i+1} \, y_i) \qquad (5)$$

Si les sommets sont orientés dans le sens antihoraire, la valeur $\Lambda (p)$ est positive, sinon elle est négative.

La surface d'un golygone est simplement obtenue en faisant la somme de celle des n polygones qui le composent.

$$\Lambda (g) = \sum_{i=0}^{n-1} \Lambda (p_i) \qquad (6)$$

Le périmètre d'un polygone est obtenu en faisant la somme des distances entre deux sommets consécutifs.

$$\Omega (p) = \sum_{i=0}^{m-1} \left[(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 \right]^{\frac{1}{2}} \qquad (7)$$

Comme pour la surface, le périmètre d'un golygone est égal à la somme des périmètres de chacune de ses composantes.

II.3.2 Inclusion d'un point

Puisqu'un golygone représente essentiellement une bipartition des points de l'espace en zones d'appartenance et de non-appartenance, il est intéressant de pouvoir connaître, pour un point particulier, s'il appartient ou non à un golygone g.

Dans la section traitant de normalisation, nous avons vu que, pour un golygone fini, les composantes situées à un niveau impair dans l'arborescence d'inclusion délimitaient vers l'intérieur les zones d'inclusion ou d'appartenance, alors que celles situées à un niveau pair délimitaient, toujours vers l'intérieur, les zones d'exclusion. Inversement, pour un golygone infini, ce sont les polygones situés aux niveaux pairs qui délimitent vers l'intérieur les zones d'appartenance. Un point $r \in \mathbf{R}^2$ sera donc inclus dans g s'il est naïvement inclus dans un nombre impair de composantes pour un g fini, et dans un nombre pair de composantes pour un g infini.

Il y a deux approches classiques pour tester l'inclusion naïve d'un point dans un polygone p. La première (Magnenat-Thalmann et Thalmann, 1983) consiste à faire la somme des angles formés par un sommet v_i, le point r, et un sommet v_{i+1}, pour tous les sommets $v_i \in p$. L'équation décrivant la fonction d'inclusion $\gamma(r, p)$ est:

$$\gamma(r, p) = \frac{1}{2\pi} \sum_{i=0}^{m-1} angle\ (v_{i+1}, r, v_i) \qquad (8)$$

La somme des angles est toujours égale à $k2\pi$. Le point r est extérieur à $|p|$ si $k = 0$. Sinon, la valeur de k indique combien de fois le polygone tourne autour de r alors que son signe donne l'orientation de p. Ainsi p est simple et positif si $k = 1$, alors qu'il est simple et négatif si $k = -1$. Si $k > 1$ ou si $k < -1$, p n'est pas simple.

Dans la seconde méthode, il s'agit de faire passer une droite horizontale par le point r et de compter le nombre de points d'intersection entre cette droite et les segments de p. Le point r est inclus dans $|p|$ si et seulement si le nombre de points d'intersection à gauche (ou à droite) de r est impair (Shamos et Preparata, 1985; Bowyer et Woodwark, 1983).

Cette approche a l'avantage d'être simple mais le désavantage de ne pas donner autant de renseignements que la première. Cependant, il y a moyen d'utiliser la simplicité des tests d'intersection pour implanter de façon plus efficace le calcul de la fonction $\gamma(r, p)$. Il s'agit non seulement de vérifier s'il y a intersection, mais aussi de donner aux vecteurs impliqués dans les points d'intersection retenus, une valeur en fonction de leur orientation. En effet, en réalisant que chaque intersection d'un vecteur avec la droite horizontale représente un regroupement des angles par portion de π, nous pouvons donner la valeur 1 aux vecteurs qui coupent la droite de haut en bas (demi-tour positif égal à une somme de π), et -1 à ceux qui la coupent de bas en haut (demi-tour négatif égal à une somme de $-\pi$). En faisant la somme de toutes ces valeurs, nous retrouvons exactement le k obtenu en utilisant le calcul des angles.

Rappelons que, pour être valide, un golygone doit être simple et normalisé. Par conséquent, aucune composante ne peut tourner plus d'une fois autour du point à tester, et l'orientation des segments impliqués dans les intersections alterne de gauche à droite, le segment le plus à gauche ayant une valeur de +1 pour un golygone fini, et de -1 pour un golygone infini. Il est donc possible d'appliquer l'approche modifiée à tous les segments d'un golygone, sans tenir compte des composantes auxquelles ils appartiennent. La figure 2.7 montre une ligne h traversant le golygone de la figure 2.5. Les segments intersectant h reçoivent une valeur qui alterne de +1 à -1 et tous les points de la droite h situés entre un segment +1 et un segment -1 sont inclus dans le golygone.

C'est d'ailleurs la même approche qui est utilisée dans la technique de *"scan-line"* servant à faire le remplissage d'un ou de plusieurs polygones (Knott, 1987). En effet, puisqu'un golygone est composé de n polygones (figures géométriques fermées), toute droite doit le couper en $2n$ points ($n \geq 0$) qui, une fois triés en valeur croissante de x, entraînent la formation de n segments de hachurage. En faisant ainsi passer un ensemble de lignes parallèles à travers un golygone normalisé fini, tous les segments générés débutent à un point d'intersection impliquant un segment allant de haut en bas et se terminent au point d'intersection suivant, impliquant un segment allant de bas en haut. Nous sommes ainsi assurés que les segments obtenus

sont bien inclus dans les régions d'appartenance délimitées par les frontières du golygone.

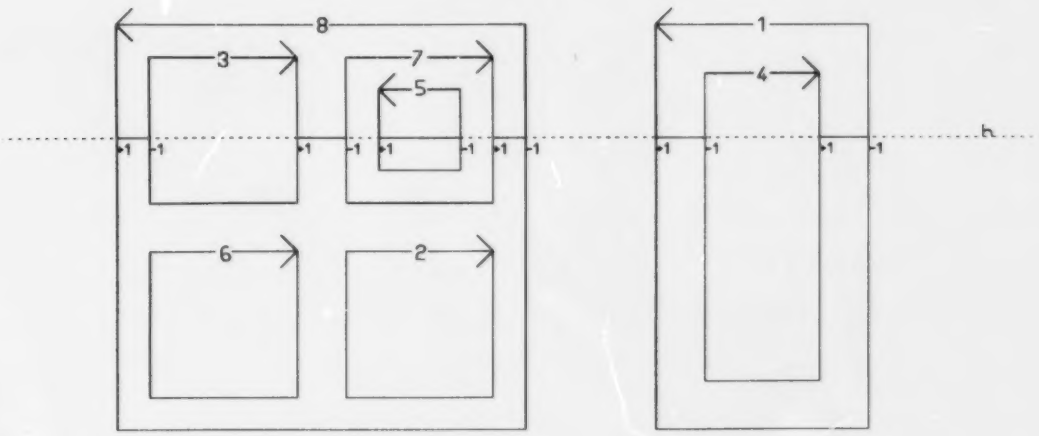


Figure 2.7 Ligne de hachurage à travers un golygone fini

II.3.3 Simplicité

Pour être réalisable, un golygone doit d'abord être simple, c'est-à-dire qu'aucune paire de segments, non consécutifs dans un même polygone, ne doit intersecter. Contrairement à ce que nous venons de voir dans les deux sections précédentes, la vérification de la simplicité d'un golygone ne se fait pas simplement en vérifiant celle de chacune de ses composantes. En effet, les segments de deux polygones simples peuvent intersecter entre eux, et alors, le golygone n'est ni simple, ni valide.

Soit μ l'ensemble de tous les segments appartenant aux n polygones du golygone g. Alors, g sera simple si, pour toute paire de segments $a, b \in \mu$, a et b non consécutifs dans le même polygone, $\delta_{(a,b)} = 0$. Soit x^a, y^a et x^b, y^b les coordonnées respectives de a et b, $\Delta x^a, \Delta y^a$ et $\Delta x^b, \Delta y^b$ les valeurs de déplacement en x et y pour ces segments. Alors, pour calculer les valeurs $\lambda_a(b)$ et $\lambda_b(a)$ nécessaires à l'évaluation de la fonction $\delta_{(a,b)}$, il suffit d'appliquer les formules de Cramer suivantes (Bowyer et Woodwark, 1983):

$$\lambda_a(b) = \frac{(y^b - y^a)\,\Delta x^b - (x^b - x^a)\,\Delta y^b}{(\Delta x^b\ \Delta y^a) - (\Delta x^a\ \Delta y^b)} \qquad (9)$$

$$\lambda_b(a) = \frac{(y^b - y^a)\,\Delta x^a - (x^b - x^a)\,\Delta y^a}{(\Delta x^b\ \Delta y^a) - (\Delta x^a\ \Delta y^b)} \,. \qquad (10)$$

Un dénominateur égal à zéro signifie que les deux segments sont parallèles. La complexité d'une implantation naïve du test de simplicité qui consiste à appliquer ces formules pour toute paire de segments est de l'ordre de $O(m(m-3)/2)$, où m est la cardinalité de l'ensemble μ, en supposant qu'on ne teste un segment ni avec lui-même ni avec ses deux voisins, et que les tests ne se font pas dans les deux sens (c'est-à-dire qu'après vérification de l'intersection de a avec b il n'est évidemment pas nécessaire de vérifier celle de b avec a). C'est donc une des opérations les plus coûteuses et qui mérite des efforts d'optimisation.

II.4 Les opérations unaires sur les golygones

II.4.1 Complémentarisation

La complémentarisation d'un golygone g sert à inclure tous les points de l'espace qui n'appartenaient pas à g et à exclure tous ceux qui lui appartenaient. C'est donc bien un complément au sens des ensembles et le nouveau golygone sera noté $\overline{g}$. Ainsi:

$$r \in \overline{g} \Longleftrightarrow r \notin g, \ \forall \ r \in \mathbf{G}$$

Cette opération se fait en complémentant chacun des polygones p constituant g.

$$\overline{g} = \{ \overline{p_0}, \overline{p_1}, \cdots, \overline{p_{n-1}} \}, \quad \forall \ p_i \in g \tag{11}$$

A son tour, le complément d'un polygone p est défini comme

$$\overline{p} = \{ v_{m-1}, v_{m-2}, \cdots, v_0 \} \tag{12}$$

c'est-à-dire qu'il est obtenu en inversant la séquence de ses m sommets et, par conséquent, la direction de ses m arêtes. La complexité est donc de l'ordre de $O(m)$.

II.4.2 Transformations linéaires

La transformation d'un golygone g, notée T(g), sert à modifier les contours des zones d'appartenance définies par g en appliquant une même transformation à tous les points de l'espace:

$$r \in g \Longleftrightarrow T(r) \in T(g), \ \forall \ r \in \mathbf{G}$$

La transformation d'un point x, y peut être exprimée par l'équation suivante:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} ax + cy + e \\ bx + dy + f \end{bmatrix} \tag{13}$$

En pratique, cette transformation n'est pas appliquée à tous les points de $\mathbf{R}^2$ mais à chacun des sommets de g, ce qui donne un golygone dont l'aspect est modifié. Des exemples de modifications classiques sont: changer l'échelle utilisée, déplacer tout le golygone par un certain vecteur de translation, ou encore le faire tourner autour d'un certain point.

Afin qu'une transformation puisse être exprimée par une seule matrice, nous pouvons adopter la méthode des *coordonnées homogènes* (Foley et Van Dam 84) où les coordonnées d'un point sont représentées par un vecteur à trois éléments. Il est alors possible d'utiliser une matrice de transformation 3×3 et l'équation exprimant cette transformation devient:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + cy + e \\ bx + dy + f \\ 1 \end{bmatrix} \qquad (14)$$

Le problème se résume donc à former cette matrice de transformation. Certains systèmes graphiques utilisent une matrice fixe correspondant à l'application, dans un ordre prédéterminé, de matrices de transformation simple. Ainsi le système *CORE* (Harrington 83) n'offre à l'utilisateur que la possibilité de faire, dans l'ordre:

1. un changement d'échelle par un facteur S_x et S_y
2. une rotation d'un angle θ autour du point (0 , 0)
3. une translation par un vecteur de déplacement T_x , T_y

et la matrice de transformation globale est:

$$\begin{bmatrix} S_x\cos\theta & -S_y\sin\theta & T_x \\ S_x\sin\theta & S_y\cos\theta & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

Cette approche n'est pas très flexible. Afin d'appliquer les transformations dans un autre ordre, ou de spécifier, pour le changement d'échelle ou la rotation, un point d'origine autre que (0, 0), il faut transformer tous les points en plusieurs étapes successives, ce qui est coûteux. De plus, les transformations moins habituelles, telles les désaxements (en anglais: *shear*) et les réflexions par rapport à une diagonale,

demandent un calcul non trivial de la séquence adéquate de rotations et de changements d'échelle à effectuer.

Cependant, puisque les matrices forment un groupe sur lequel les opérations sont associatives, il est possible d'appliquer une suite de transformations simples, non pas aux points, mais directement à une matrice, afin de créer n'importe quelle matrice de transformation et d'appliquer cette matrice une seule fois à l'ensemble des points à transformer. Chaque matrice intermédiaire est obtenue en pré-multipliant la matrice courante par la matrice de transformation correspondant à la prochaine transformation dans la séquence désirée:

$$\begin{bmatrix} a' & c' & e' \\ b' & d' & f' \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} aa'+bc' & ca'+dc' & ea'+fc'+e' \\ ab'+bd' & cb'+dd' & eb'+fd'+f' \\ 0 & 0 & 1 \end{bmatrix} \quad (15)$$

On obtient de cette façon une nouvelle matrice combinant toutes les transformations déjà spécifiées. La valeur de chacune des variables a', b', c', d', e' et f' dépend du type de transformation et, dans le cas où une transformation est fonction du point d'origine choisi, ces valeurs doivent refléter la transformation autour de l'origine courante.

Au départ, le point d'origine courant est (0 , 0) et la matrice de transformation correspond à la matrice d'identité:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Nous pouvons maintenant décrire les matrices correspondant aux diverses opérations.

- Translation par un vecteur de déplacement T_x T_y.

$$\begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Changement d'échelle par un vecteur de déplacement $S_x \ S_y$ avec x_o, y_o comme point d'origine.

$$\begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} S_x & 0 & -x_o S_x + x_o \\ 0 & S_y & -y_o S_y + y_o \\ 0 & 0 & 1 \end{bmatrix}$$

- Rotation d'un angle θ autour du point $(x_o \ , y_o)$;

$$\begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & -x_o\cos\theta + y_o\sin\theta + x_o \\ \sin\theta & \cos\theta & -x_o\sin\theta - y_o\cos\theta + y_o \\ 0 & 0 & 1 \end{bmatrix}$$

- Désaxement d'une valeur H_x vers l'axe x avec origine $(x_o \ , y_o)$; †

$$\begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & H_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & H_x & -y_o H_x \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Désaxement d'une valeur H_y vers l'axe y avec origine $(x_o \ , y_o)$;

$$\begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ H_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ H_y & 1 & -x_o H_y \\ 0 & 0 & 1 \end{bmatrix}$$

† Les formules exprimant le désaxement peuvent différer selon qu'on parle de désaxement vers l'axe (Harrington) ou de désaxement de l'axe (Van Dam et Foley).

On note que le désaxement vers l'axe x ne dépend que de y_o alors que celui vers l'axe y ne dépend que de x_o.

Signalons enfin que la complexité est de l'ordre de $O(m)$ pour une combinaison quelconque de transformations linéaires. En effet, la complexité n'augmente pas avec le nombre de transformations puisque celles-ci sont faites sur la matrice de transformation et non pas directement sur les coordonnées.

II.5 Les opérations binaires sur les golygones

Une opération binaire sur deux golygones g^1 et g^2 donne un nouveau golygone g pour lequel l'état d'appartenance de chacun des points de l'espace bidimensionnel correspond au résultat d'une opération logique sur deux variables binaires. La première correspond à l'état d'appartenance d'un point par rapport à g^1 et la seconde, à celui du même point par rapport à g^2:

$$r \in (g^1 \,\mathbf{op}\, g^2) \iff (r \in g^1)\,\mathbf{op}\,(r \in g^2),\ \forall\ r \in \mathbf{G}$$

Dans la suite de cette section, les opérations binaires que nous allons considérer sont l'**union**, qui correspond à l'opération logique "OU", et l'**intersection**, qui correspond à l'opération logique "ET" .

Nous savons que si on applique l'une de ces opérations sur deux polygones, le résultat n'est pas toujours fermé sur la classe des polygones et peut donner un golygone. Nous voulons définir directement ces opérations sur les golygones et obtenir une classe d'objets qui nous assure que tout résultat peut être manipulé de la même façon.

Avant de présenter la méthode que nous avons développée, nous allons énoncer quelques propriétés de base qui sont toujours satisfaites.

II.5.1 Propriétés des opérations binaires

La première propriété concerne la relation qui existe entre les surfaces des golygones sur lesquels nous désirons effectuer une opération binaire, et les surfaces des résultats pour chacune des deux opérations. Nous savons que la surface d'un golygone correspond à l'intégrale sur tous les points appartenant à ce golygone. Soit A l'ensemble de tous les points appartenant à g^1, B l'ensemble de tous les points appartenant à g^2 et U l'ensemble universel $\mathbf{R}^2$. Alors, en appliquant la définition mathématique de l'union de deux ensembles (Stoll, 1963) nous avons que:

$$A \cup B = A + (B \cap \bar{A}) = A + (B \cap (U - A)) = A + B - (B \cap A) \quad (16)$$

Notons $\Lambda (g^1)$ et $\Lambda (g^2)$ les surfaces respectives des golygones g^1 et g^2. En remplaçant, dans l'équation précédente, A par $\Lambda (g^1)$ et B par $\Lambda (g^2)$ et en faisant passer certains termes de l'autre côté du signe d'égalité, l'équation suivante est vérifiée:

$$\Lambda (g^1) + \Lambda (g^2) = \Lambda (g^1 \cup g^2) + \Lambda (g^1 \cap g^2) \qquad (17)$$

Soit maintenant $\Omega (g)$ le périmètre d'un golygone g. Alors, la relation qui existe entre le périmètre des golygones de départ et celui des golygones qui résultent de l'union et de l'intersection s'exprime par une équation semblable à celle exprimant la relation entre les surfaces.

$$\Omega (g^1) + \Omega (g^2) = \Omega (g^1 \cup g^2) + \Omega (g^1 \cap g^2) \qquad (18)$$

En effet, le périmètre d'un golygone est égal à la somme des longueurs des segments formant sa frontière et la frontière résultant d'une opération binaire entre deux golygones ne peut qu'être formée de segments appartenant à g^1 ou à g^2. Il reste à prouver que tout point de frontière appartient soit à l'union, soit à l'intersection.

Soit une portion de frontière n'intersectant avec aucune autre. Elle délimite une région d'appartenance d'un côté et une de non-appartenance de l'autre côté. Considérons maintenant, sans perte de généralité, que cette portion de frontière appartient au golygone g^1 et examinons son état d'appartenance par rapport au golygone g^2. Puisque, par hypothèse, cette portion de frontière n'intersecte avec aucune autre, elle est naïvement incluse, soit dans une région d'appartenance induite par g^2 (figure 2.8 A), soit dans une région de non-appartenance (figure 2.8 B). Dans le premier cas, elle ne peut border la zone d'union mais elle borde la zone d'intersection des régions d'appartenance induites par les deux golygones. Dans le second cas, on a la situation inverse: la portion de frontière ne peut border une zone d'intersection puisqu'elle est située dans une région de non-appartenance par rapport à g^2, mais elle doit border une zone d'union.

En appliquant le même raisonnement à toute portion de frontière appartenant à g^1 et à g^2, on peut conclure que chacune contribue, soit à la frontière de l'union, soit

à celle de l'intersection, mais jamais aux deux à la fois. Notons que le raisonnement tient même lorsqu'il y a intersection entre les frontières, en considérant séparément les portions situées de part et d'autre de l'intersection. Quant aux points d'intersection eux-mêmes, ils impliquent chacun un point appartenant à la frontière de g^1 et un point appartenant à celle de g^2. Alors, puisque les figures formées sont fermées, si on suppose qu'un de ces points participe à la frontière de l'union, l'autre participera à celle de l'intersection. Ce qui nous amène à présenter la dernière propriété.

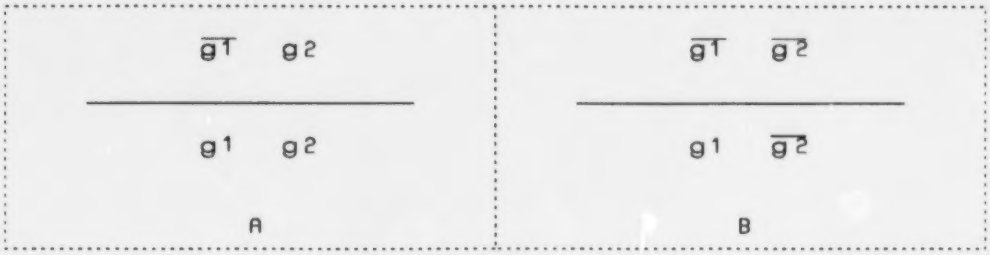


Figure 2.8 Régions d'appartenance de chaque côté d'une ligne frontière

Soit $V\ (\ g\ )$ le nombre de sommets dans un golygone g, et X le nombre de points d'intersection entre g^1 et g^2. Alors:

$$V\ (g^1) + V\ (g^2) = V\ (g^1 \cup g^2) + V\ (g^1 \cap g^2) - 2X \qquad (19)$$

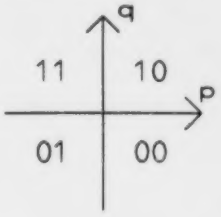
Nous venons de voir que tous les segments n'intersectant pas font partie soit du résultat de l'union, soit de celui de l'intersection. Les sommets impliqués dans ces segments participent donc eux aussi, soit à l'intersection, soit à l'union. Par contre, lorsque deux segments intersectent, le point d'intersection est un point de décision sur la portion de segment à garder pour l'union, et celle à garder pour l'intersection. Ce point constitue donc un nouveau sommet participant aux golygones résultant des deux opérations.

Les deux dernières propriétés, soit celle sur le périmètre et celle sur les sommets, constituent la motivation principale qui a guidé l'élaboration de l'algorithme servant à obtenir l'union et l'intersection de deux golygones.

II.5.2 Algorithmes

L'algorithme permettant de faire l'union ou l'intersection de deux polygones, non f-disjoints, consiste essentiellement à suivre les frontières et à prendre les bons virages aux croisements (points d'intersection). Le problème se résume à déterminer quelle direction prendre lorsqu'on visite un point d'intersection.

Supposons le segment $a \in p$ et le segment $b \in q$ et soit $\delta_{(a,b)} = 1$. Alors, l'intersection entre a et b délimite quatre régions d'appartenance telles qu'illustrées à la figure 2.9.



00 : points n'appartenant ni à p ni à q
01 : points appartenant à q mais pas à p
10 : points appartenant à p mais pas à q
11 : points appartenant à p et à q

Figure 2.9 Régions d'appartenance définies par l'intersection de deux segments

Longer la frontière de l'intersection consiste à prendre les portions de segments bordant la région 11 c'est-à-dire, à chaque point d'intersection, emprunter le segment sortant *à gauche*, alors que longer celle de l'union consiste à prendre les portions de segments qui bordent la région 00, donc emprunter, à chaque intersection, le segment *à droite*. Puisque cette règle de décision s'appuie sur une condition locale, elle s'applique quel que soit le signe des polygones impliqués.

Des algorithmes basés sur ce principe ont déjà été présentés dans le cas de l'intersection de deux polygones convexes (O'Rourke, Chien, Olson et Naddor, 1982), et dans celui de deux polygones simples quelconques (Von Hohenbalken, 1985). En utilisant des polygones d'entrée convexes, O'Rourke et coll. évitent le problème de générer plus d'un polygone résultant. Puisque le résultat obtenu est lui aussi un polygone convexe, il est alors possible d'appliquer séquentiellement le même algorithme pour faire l'intersection de plusieurs polygones. Quant à Von Hohenbalken, il reconnait que le résultat de l'intersection de deux polygones non convexes peut donner un objet qui n'est pas du même type que celui des objets utilisés en

entrée, et il donne une définition généralisée du polygone. Il ne précise toutefois pas comment ce polygone généralisé peut être réutilisé en entrée pour l'algorithme qu'il propose.

L'approche que nous venons de décrire fonctionne lorsqu'au moins un segment de p intersecte avec un segment de q. Par contre, le cas général impliquant deux polygones f-disjoints n'est pas, à notre connaissance, traité dans la littérature. Von Hohenbalken impose à p et à q d'être positifs. Alors, l'intersection de p et q donne:

1 - p, si p est imbriqué dans q;
2 - q, si q est imbriqué dans p;
3 - $\varnothing$, si p et q sont séparés.

Cependant, ce résultat n'est pas correct, au sens des ensembles de points inclus par p et q, si p et q sont de signes contraires, ou s'ils sont tous les deux négatifs.

Nous posons maintenant la question suivante. Comment appliquer à deux golygones, les algorithmes qui existent déjà pour faire l'intersection de deux polygones? Nous n'avons trouvé aucune référence qui soulève cette question. Pourtant, il n'est ni efficace, ni trivial, de décomposer une opération binaire entre deux golygones en une suite d'opérations binaires entre les polygones composantes. En effet, une composante du résultat ne dépend pas seulement de deux polygones mais peut intégrer des portions de frontières de plusieurs polygones, comme le montre l'exemple de la figure 2.10. Les polygones 1, 2, 3 et 4 composent le golygone g^1 alors que le golygone g^2 se résume à la composante 5. Les régions pointillées représentent le résultat de l'intersection. Ce résultat étant composé de deux régions d'appartenance disjointes, il est impossible de l'obtenir seulement par une combinaison d'intersections entre les différentes composantes des golygones impliqués. Eventuellement, il faudra au moins faire l'union de résultats intermédiaires.

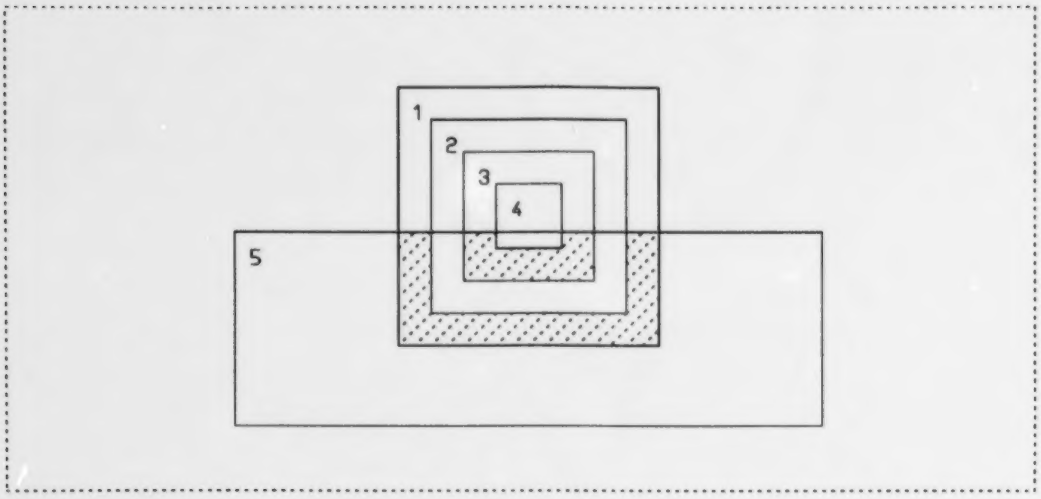


Figure 2.10 Intersection impliquant plus de deux polygones par composante

De plus, lorsque des composantes d'un golygone sont f-disjointes de l'autre, le résultat ne dépend pas seulement d'une intersection de frontières mais aussi de la relation d'inclusion qui existe entre les composantes, donc de la structure arborescente. La figure 2.11 illustre un cas très simple. Le golygone g^1 est composé d'un polygone positif (1) et d'un polygone négatif (2), alors que le polygone positif 3 appartient à g^2. Supposons que l'on veuille décomposer l'union de g^1 et g^2 en impliquant les composantes deux à deux et que l'on commence par faire l'union des composantes 2 et 3. Le résultat intermédiaire obtenu est $\mathbf{R}^2$. Alors, l'union avec toute autre composante ne peut donner que $\mathbf{R}^2$. Par conséquent, pour avoir le bon résultat, il est nécessaire de faire appel à une opération d'intersection.

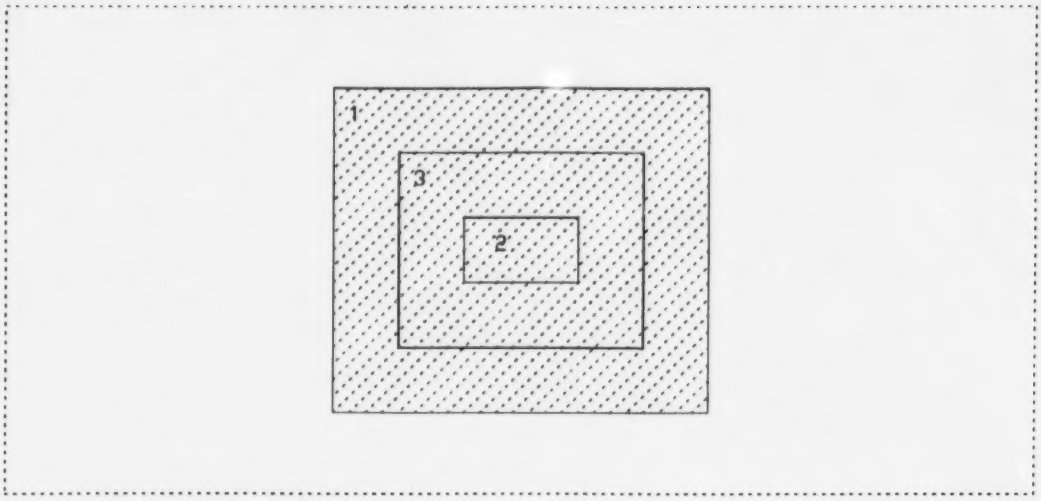


Figure 2.11 Union impliquant la relation d'inclusion entre les composantes

Nous affirmons donc que faire les opérations par paires implique la connaissance des deux *expressions booléennes caractéristiques* des golygones de départ. Il est ensuite possible, en appliquant à ces deux expressions l'opération logique désirée et, après simplification à l'aide des lois de deMorgan de l'expression obtenue, de trouver la combinaison adéquate d'opérations. De plus, un pré-requis pour trouver ces expressions caractéristiques est de connaître l'arborescence de chacun des deux golygones impliqués.

Dans ce qui suit, nous montrons que cette dernière approche est non seulement inefficace mais aussi inutile, et nous proposons plutôt une méthode pour calculer directement l'intersection ou l'union de deux golygones.

Soit g^1 et g^2 les deux golygones simples et normalisés sur lesquels nous désirons effectuer une opération binaire et supposons que, sur chaque composante polygonale, il y a des points d'intersection. Alors, deux observations s'imposent lorsqu'on suit les frontières. Premièrement, tous les sommets rencontrés entre deux points d'intersection consécutifs appartiennent au même golygone. Deuxièmement, puisque les golygones sont simples, tout point d'intersection implique nécessairement un segment appartenant à g^1 et un segment appartenant à g^2. Par conséquent,

comme dans le cas de l'intersection entre deux polygones, toute décision de quitter le golygone courant ne peut se faire qu'à un point d'intersection.

Ces observations nous amènent à penser que l'approche utilisée pour faire l'intersection de deux polygones peut tout aussi bien être utilisée dans le cas de deux golygones, du moins lorsque toutes les composantes intersectent avec l'autre golygone. L'algorithme 3 sert à trouver les frontières du golygone résultant d'une opération binaire:

Algorithme 3:

ETAPE 1: trouver tous les points d'intersection entre g^1 et g^2;
initialiser ces points d'intersection à *non visité*

ETAPE 2: suivre les frontières de g^1 et se placer au premier point d'intersection non visité;
s'il n'y en a plus alors ARRETER

ETAPE 3: marquer le point d'intersection *visité* et le mettre dans le résultat

ETAPE 4: si on veut l'union
    suivre le segment continuant vers la droite
sinon
    suivre le segment continuant vers la gauche

ETAPE 5: poser, comme golygone courant, celui auquel appartient le segment emprunté et continuer sur ce golygone en mettant dans le résultat tous les sommets rencontrés jusqu'au prochain point d'intersection

ETAPE 6: si le point d'intersection a déjà été visité
    aller à l'étape 2
sinon
    aller à l'étape 3

Il est important de remarquer que l'algorithme peut commencer sur n'importe quel segment de n'importe quel golygone. Cependant, à cause de la propriété qui assure que tout point d'intersection fait partie de l'union et de l'intersection, tout cycle **actif**, c'est-à-dire pendant lequel tous les sommets visités font partie du résultat, démarre à un point d'intersection (étape 2) et se termine au même point. Par conséquent, à chaque cycle actif correspond une composante du résultat. Entre deux

cycles actifs, il y a une promenade passive le long des frontières servant à se positionner au prochain point d'intersection non visité. L'étape 2 assure que tous les segments et toutes les intersections ont été parcourues.

L'algorithme 3 est valide autant pour l'intersection que pour l'union, la seule différence étant dans la direction empruntée à un point d'intersection. Quoique cette observation puisse paraître évidente, elle n'est pas toujours appliquée. Par exemple, l'algorithme de Hohenbalken (1985), ne sert qu'à faire l'intersection. Pour obtenir le résultat d'une autre opération binaire, il utilise le fait que toutes les opérations sur les ensembles peuvent être exprimées en termes d'intersection et de complémentarisation. En particulier, il obtient l'union de p et q en appliquant le théorème de deMorgan:

$$\overline{p \cup q} = \overline{p} \cap \overline{q} .$$

L'algorithme que nous venons de voir donne le résultat attendu si et seulement si toutes les composantes de g^1 intersectent au moins une composante de g^2, et vice-versa. Examinons maintenant ce qui se passe lorsqu'il n'y a aucun point d'intersection entre g^1 et g^2. Cela implique qu'il n'y a que des composantes entières qui puissent faire partie du résultat d'une opération binaire, puisqu'il n'y a aucun point d'intersection où il soit possible de passer de la frontière d'une composante à celle d'une autre. De plus, par la propriété sur le périmètre, nous savons que toute composante appartient soit à l'union, soit à l'intersection. Le golygone résultant sera donc formé d'un sous-ensemble de g^1 et d'un sous-ensemble de g^2 et l'algorithme consiste à déterminer, pour chacun des polygones composant g^1 et g^2, s'il participe au résultat de l'union ou de l'intersection.

Les composantes qui entrent dans le résultat doivent être choisies en appliquant le même raisonnement que pour l'algorithme 3. Celles qui bordent les régions 00 forment l'union alors que pour l'intersection il faut prendre celles qui bordent les régions 11. Cette approche conceptuelle peut-elle être traduite directement dans un algorithme tel que ce fut le cas pour la procédure précédente? De prime abord, ça ne semble pas évident. Lorsqu'il y a une intersection, la décision à prendre localement est totalement indépendante de tout segment ne participant pas à cette intersection.

Par contre, lorsqu'une composante est f-disjointe, il faut tenir compte de son orientation et de toutes les composantes de son voisinage immédiat, c'est-à-dire, parmi les composantes de g^1 ou de g^2,

a) celles qu'elle englobe immédiatement (les enfants);

b) celle qui l'englobe immédiatement (le parent);

c) celles qui ont le même parent (les frères).

La meilleure façon de mettre ces relations en évidence est de construire l'arborescence d'inclusion de toutes les composantes de g^1 et de g^2. Il s'agit ensuite d'examiner les régions d'appartenance induites par la nouvelle arborescence formée. Dans cette arborescence, deux composantes sont dites consécutives si une englobe immédiatement l'autre. Puisque g^1 et g^2 sont, par hypothèse, normalisés, deux composantes consécutives appartenant au même golygone de départ, soit participent toutes les deux au résultat de l'opération binaire, soit n'y participent pas, et cela dépendra de leur relation avec les composantes de l'autre golygone.

Lorsque deux composantes consécutives appartiennent, l'une à g^1, l'autre à g^2, le choix des composantes dépend de leur orientation. La figure 2.12 illustre les diverses combinaisons possibles. Les lignes continues et brisées servent à distinguer un polygone p appartenant à g^1 d'un polygone q appartenant à g^2. La signification des valeurs accordées à chaque région est la même que celle définie à la figure 2.9. Remarquons que deux composantes f-disjointes n'induisent pas la formation de quatre régions d'appartenance distinctes, comme dans le cas d'une intersection, mais de trois. L'absence de région 11 signifie une intersection vide, alors que l'absence de région 00 implique une union égale à $\mathbf{R}^2$. Quant à la figure 2.13, elle aide à mieux visualiser les zones d'union et d'intersection correspondant à chacune des situations présentées à la figure 2.12, les régions d'appartenance définies par p étant hachurées par des segments dont la pente est positive, et celles définies par q par des segments à pente négative.

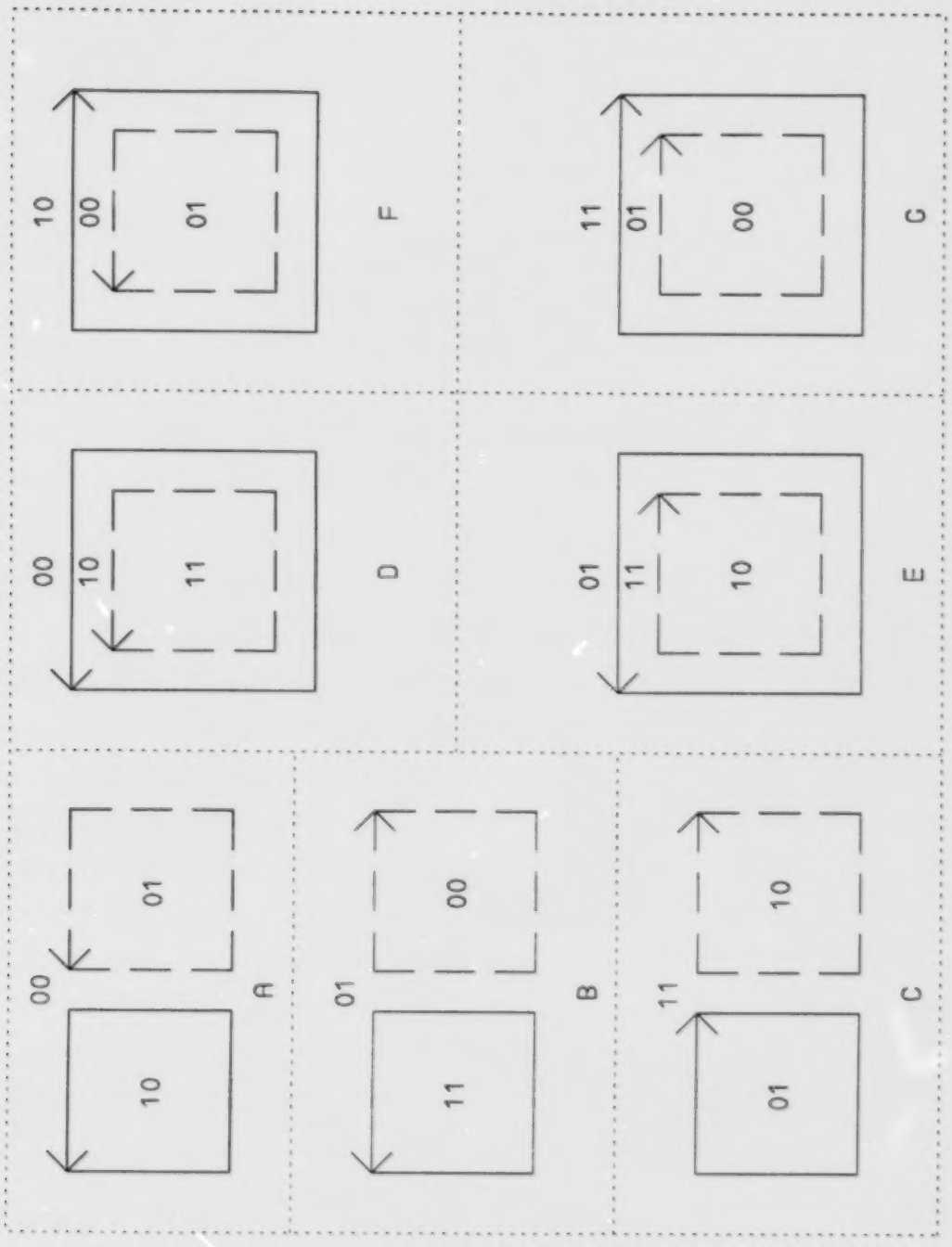


Figure 2.12 Régions d'appartenance induites par diverses combinaisons

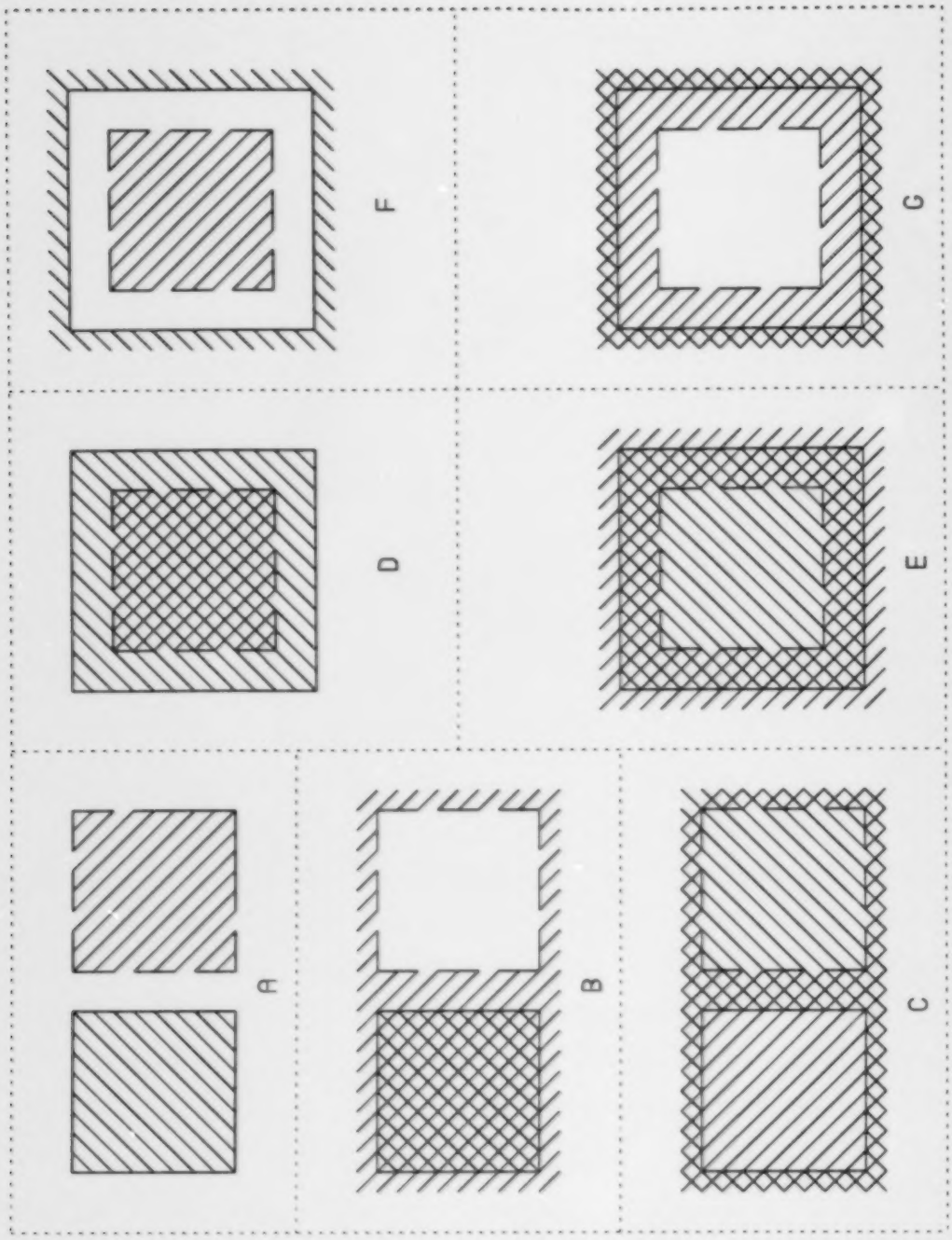


Figure 2.13 Visualisation des zones d'union et d'intersection par hachurage

A partir de toutes ces observations et après un certain nombre de recherches et de reformulations du problème, nous sommes arrivés à la conclusion qu'il existait une façon simple et élégante de choisir les composantes f-disjointes participant au résultat de l'opération binaire voulue. Celle-ci se résume dans le théorème suivant:

Théorème: Soit g^1 et g^2 deux golygones normalisés ne partageant aucun point d'intersection. Alors p, composante d'un golygone, participe à la frontière de l'intersection entre g^1 et g^2 si sa frontière est incluse dans l'autre golygone. Sinon, elle participe à la frontière de l'union.

Preuve: Pour faire la preuve de ce théorème nous nous baserons sur les propriétés présentées à la section précédente.

1] L'équation (19) assure que tout point de frontière qui n'est pas un point d'intersection appartient, soit à l'union soit à l'intersection. Il doit appartenir à l'intersection s'il est inclus dans l'autre golygone et par conséquent, il appartient à l'union s'il n'est pas inclus.

2] Le résultat de l'opération doit être un golygone et, par l'équation (18), nous savons que les frontières de ce golygone sont uniquement constituées de parties de frontières de g^1 et g^2. Supposons, sans perte de généralité, qu'un point de frontière appartient au résultat de l'union. Alors, puisque toutes les composantes sont f-disjointes, le seul cycle que l'on puisse former qui passe par ce point est la composante sur la frontière de laquelle se trouve ce point. Par conséquent, la composante entière fait partie de l'union. Le même raisonnement s'applique si un point de frontière appartient à l'intersection. On conclue que toute composante de g^1 et de g^2 est, soit entièrement dans le résultat de l'union, soit entièrement dans celui de l'intersection, et il suffit de tester l'inclusion d'un seul point d'une composante dans l'autre golygone pour décider si cette composante participe à l'union ou à l'intersection. □

Nous avons déjà dit, à la section II.3.2, que le test d'inclusion d'un point dans un golygone dépend de son niveau d'inclusion naïve dans l'arborescence correspondant à ce golygone, c'est-à-dire du nombre de composantes englobantes, et du fait que ce golygone est fini ou non. Le tableau II résume les règles du test d'inclusion.

nombre de composantes englobantes	type de golygone	résultat du test
impair	fini	inclus
impair	infini	non inclus
pair	fini	non inclus
pair	infini	inclus

Tableau II Règles du test d'inclusion d'un point dans un golygone

Nous avons maintenant des critères de sélection sur lesquels nous pouvons nous appuyer pour établir la procédure à suivre afin d'obtenir l'union ou l'intersection de deux golygones f-disjoints. L'algorithme 4 décrit cette procédure:

Algorithme 4:

ETAPE 1: pour tout $p \in g^1$ faire
si (p est inclus dans un nombre impair de composantes de g^2) alors
si (g^2 est fini) alors
p est une composante du golygone intersection
sinon
p est une composante du golygone union
sinon
si (g^2 est fini) alors
p est une composante du golygone union
sinon
p est une composante du golygone intersection

ETAPE 2: pour tout $q \in g^2$ faire
si (q est inclus dans un nombre impair de composantes de g^1) alors
si (g^1 est fini) alors
q est une composante du golygone intersection
sinon
q est une composante du golygone union
sinon
si (g^1 est fini) alors
q est une composante du golygone union
sinon
q est une composante du golygone intersection

En combinant les deux algorithmes précédents, il est possible d'obtenir le résultat voulu pour g^1 et g^2 quelconques.

Algorithme 5:

ETAPE 1: appliquer l'algorithme 3.

ETAPE 2: pour tout $p \in g^1$ tel que p est f-disjoint de g^2 faire
appliquer les critères de sélection décrits à
l'étape 1 de l'algorithme 4

ETAPE 3: pour tout $p \in g^2$ tel que p est f-disjoint de g^1 faire
appliquer les critères de sélection décrits à
l'étape 2 de l'algorithme 4

Cet algorithme nous donne le résultat voulu puisque:

- Le résultat obtenu est un bien un golygone. En effet, chaque composante est un polygone, soit formé par un cycle initié par un point d'intersection, soit correspondant à une composante polygonale de g^1 ou g^2.

- On peut trouver toutes les composantes. Si elles contiennent un point d'intersection, elles seront trouvées puisque l'algorithme balaie les segments tant qu'il reste des points d'intersection non visités. Sinon, elles seront trouvées puisque toutes les composantes f-disjointes sont examinées afin de décider de leur appartenance au résultat et que, par notre théorème, cette décision est la bonne.

- On ne trouve les composantes qu'une seule fois. Lorsqu'une nouvelle composante contenant un point d'intersection est construite, tous les points d'intersection sont marqués dès qu'ils sont visités et un point visité n'est plus considéré par la suite. Dans le cas des composantes distinctes, on ne teste leur appartenance au résultat qu'une seule fois.

En résumé, il est possible de calculer directement le résultat d'une opération binaire entre deux golygones, sans la décomposer en opérations plus simples n'impliquant que deux polygones à la fois. Le résultat obtenu est un nouveau golygone dont chaque composante, si elle contient des points d'intersection, peut intégrer des morceaux de frontière de plusieurs polygones de départ et, si elle ne dépend pas d'une intersection de frontières, dépend de la structure arborescente des golygones de départ.

III. LES ASPECTS INFORMATIQUES

Le but de ce chapitre est de présenter les principes informatiques utilisés lors de notre implantation des notions théoriques développées dans le chapitre précédent.

Les procédures que nous avons implantées peuvent être classées en trois catégories selon leur niveau d'utilisation. Au plus haut niveau, il y a tous les programmes directement accessibles à un utilisateur, alors que, au second niveau, les routines accessibles au programmeur qui désire développer ses propres utilitaires sont rassemblées sous forme de bibliothèque. Enfin, au plus bas niveau, se trouvent les primitives, elles aussi intégrées à la bibliothèque, mais qui ne sont appellées que par les programmes ou les routines du second niveau. Ces dernières ne sont donc accessibles ni à l'utilisateur, ni au programmeur.

Dans ce chapitre, nous définissons d'abord le format externe des fichiers d'entrée et l'organisation interne des données, puis nous décrivons les principaux détails d'implantation des diverses procédures et nous traitons de quelques problèmes de dégénérescence. Enfin nous montrons comment utiliser nos programmes.

Mentionnons que, dans les pages qui suivent, tout mot écrit en caractère gras sert à introduire une nouvelle définition ou à mettre en évidence la première occurence d'un nom de variable, routine ou programme. Toute autre référence au même mot est en italique.

III.1 Les structures de données

La plupart de nos programmes sont des filtres qui reçoivent des golygones en entrée et donnent des golygones en sortie. Pour cette raison, il est essentiel que la représentation externe des données soit spécifiée selon des règles très précises. C'est l'aspect qui sera traité dans la prochaine section.

Pour ce qui est du traitement informatique, on a besoin de structures qui permettent un accès efficace aux informations et qui minimisent l'espace utilisé et le temps de calcul, toutes considérations qui suggèrent l'utilisation des grandes possibilités du

langage C. Cet aspect sera traité dans la section consacrée à la représentation interne des données.

III.1.1 Le format externe

Un format externe permet à toutes les commandes qui manipulent les golygones d'être compatibles au niveau entrée/sortie. Lorsqu'il est possible de passer directement la sortie d'une commande à l'entrée d'une autre, ces commandes sont appelées **filtres**. L'utilisation en série des *filtres* permet de former des expressions plus complexes au niveau du système d'exploitation et d'éviter ainsi la création de fichiers intermédiaires de résultats.

Le format externe utilisé pour décrire les golygones reflète le fait qu'un golygone est, tout d'abord, un ensemble de polygones. La première ligne est une chaîne de caractères, possiblement vide, servant à identifier le golygone. Tous les caractères d'imprimerie disponibles sont permis, sauf le caractère *blanc* et le caractère de tabulation. Deux caractères ont une signification spéciale s'ils se trouvent en début (premier caractère) de ligne. Le premier est le "%", qui marque le début d'une composante polygonale. Le second est le "$", et il marque la fin du golygone. Les lignes situées entre deux "%" ou entre un "%" et un "$" servent à la description d'un élément de l'ensemble (polygone) et chacune contient les coordonnées cartésiennes d'un sommet. Il faut remarquer que le premier sommet n'a pas à être répété pour marquer la fermeture du polygone puisque les caractères spéciaux (% et $) l'indiquent implicitement. Toute autre information provoque une erreur, si elle se trouve en début de ligne, ou est ignorée, si elle suit une information pertinente sur la même ligne. Ce dernier cas permet, à l'utilisateur qui le désire, d'ajouter des informations additionnelles qui ne seront pas nécessairement traitées ni préservées par les filtres. Ainsi, toute information additionnelle concernant une composante peut être ajoutée sur la ligne contenant le "%" marquant le début de cette composante, alors qu'une information suivant des coordonnées s'applique au segment dont la queue est définie par ces coordonnées. Par exemple, la description externe du golygone de la figure 2.1A est:

FIGURE–2.1

```
% Composante finie
20 20     Coin inférieur gauche
60 20
60 60     Coin supérieur droit
20 60
% Composante imbriquée dans la première
24 40
40 24
56 40
40 56
$
```

Cette représentation est très souple et peut même s'adapter aux cas particuliers de l'ensemble vide (∅) et de l'ensemble universel (dans notre cas, $\mathbf{R}^2$). On considère que l'ensemble universel est un golygone à une composante et le fichier de description contient trois lignes: une pour le nom, une pour le caractère de début de composante et une dernière pour le caractère de fin de golygone.

```
Univers_bidimensionnel
%  —> 1 composante sans segment
$
```

Quant à l'ensemble vide, c'est un golygone ne contenant aucune composante donc le fichier ne contiendra qu'un nom en première ligne et le caractère "$" en deuxième (et dernière) ligne.

```
Golygone_Vide
$  —> aucune composante
```

III.1.2 Le format interne

Pour être efficace, l'organisation interne des données doit minimiser l'utilisation de l'espace mémoire, non seulement en évitant le plus possible la redondance dans les informations, mais aussi en n'allouant que l'espace nécessaire pour stocker données et résultats intermédiaires. Elle doit aussi permettre d'accéder rapidement à toutes les informations et de les transmettre facilement d'une routine à l'autre.

Le langage C (Kernighan et Ritchie, 1978) apparaît comme très approprié pour satisfaire ces critères. Entre autres possibilités, il permet de modifier l'allocation de l'espace mémoire (fonction **realloc**), de définir des structures et de donner comme paramètres aux routines des pointeurs vers ces structures.

Les informations de base sont réparties dans quatre types de structure: **golygone**, **polygone**, **segment** et **point d'intersection**. La structure décrivant le golygone est codée comme suit:

```
typedef struct golygon {
        char name[NAMESIZE];
        int nbseg;
        int nbcomp;
        double area;
        COMP *root;
        COMP *comp;
        SEGMENT *seg;
} GOLYGON;
```

La variable *name* contient l'information lue dans la première ligne du fichier d'entrée. Les variables *nbseg* et *nbcomp* contiennent respectivement le nombre total de segments et de composantes polygonales dans ce golygone, alors que la valeur calculée de la surface du golygone est emmagasinée dans *area*. La variable *root* est un pointeur vers une composante bidon représentant tout l'univers cartésien en deux dimensions. Elle doit être égale au pointeur nul tant que l'arborescence d'inclusion n'est pas montée. Lorsque cette arborescence est bâtie, *root* pointe vers la première composante qui n'est imbriquée dans aucune autre.

Les deux derniers champs contiennent des pointeurs vers des tableaux de structures: *comp* est un pointeur vers la première composante du golygone tandis que *seg* pointe vers le premier segment.

Tous les polygones composant un golygone sont placés consécutivement en mémoire dans un tableau dont chaque élément correspond à une structure de type *COMP*.

```
#define SHARE  1
#define CLOCKWISE  2
typedef struct comp {
        char flag;
        int nbseg;
        int nbcomp;
        double area;
        COMP *left;
        COMP *right;
        SEGMENT *seg;
} COMP;
```

Cette structure est très semblable à celle décrivant un golygone. Le premier champ contient la variable *flag* qui sert à stocker de l'information interne pendant le traitement. Présentement, les deux premiers bits sont utilisés lors des opérations binaires. Cette variable est d'abord initialisée à zéro (0). La constante *SHARE* sert à ''allumer'' le bit 0 si la composante partage au moins un point d'intersection avec une autre composante alors que *CLOCKWISE* sert à allumer le bit 1 si le polygone tourne dans le sens horaire. Par défaut, une composante est donc considérée comme f-disjointe et d'orientation antihoraire. Le nombre de segments formant le polygone est conservé dans la variable *nbseg*. Un polygone étant un synonyme d'**une** composante polygonale, la signification de *nbcomp* ne peut évidemment pas être la même que dans le golygone. Cette variable sert maintenant à garder le nombre de composantes englobant complètement le polygone, alors que *left* pointe vers le premier polygone englobé (aîné) et *right* pointe vers un polygone situé au même niveau dans l'arborescence d'inclusion (cadet). Enfin, *seg* est un pointeur vers le premier segment formant le polygone.

Tous les segments composant un golygone sont, eux aussi, placés consécutivement en mémoire dans un tableau dont chaque élément correspond à une structure de type *SEGMENT*.

```
typedef struct segment {
        double x;
        double y;
        double dx;
        double dy;
        int nxt;
        CROSS *cross1;
} SEGMENT;
```

Les informations contenues dans ce tableau sont accessibles tant au golygone qu'à ses composantes. Les valeurs x et y correspondent aux coordonnées cartésiennes du sommet **queue** d'un segment. Les valeurs dx et dy indiquent le déplacement pour obtenir les coordonnées de la **tête** du segment.

L'organisation de tous les segments en tableau fournit un moyen rapide et efficace d'accéder à tous les segments d'un golygone sans nécessairement s'occuper du polygone auquel appartient chacun d'eux. Nous parlerons de *prochain segment* pour référer à un segment qui en suit immédiatement un autre dans le tableau des segments, alors que l'expression *segment suivant* désignera plutôt le segment qui suit sur la frontière d'un polygone, c'est-à-dire celui qui est incident à la tête du segment courant. Le pointeur *seg* de la structure *golygone* permet de se positionner au début du tableau et de passer d'un segment à l'autre en incrémentant toujours le pointeur de la valeur +1.

La variable *nxt* joue un rôle lorsqu'il s'agit d'accéder aux seuls segments d'un polygone particulier. Elle contient une valeur d'incrément qui, ajoutée au pointeur vers le segment courant, donnera la position du segment suivant dans le polygone. Ainsi, pour un polygone contenant m segments, la variable *nxt* aura la valeur +1 pour les $(m-1)$ premiers segments et la valeur $-(m-1)$ pour le dernier. Le pointeur *seg* de la structure *polygone* permet de se positionner à l'élément du tableau correspondant au premier segment de ce polygone. Il suffit ensuite de suivre les

segments un à un, jusqu'à ce que *nxt* contienne une valeur inférieure à 1. A ce moment, on sait qu'on a atteint le dernier sommet du polygone. Dans le cas dégénéré d'un polygone réduit à un seul point, la variable *nxt* est égale à zéro. Cette organisation permet aussi de suivre la frontière d'un polygone à partir de n'importe lequel de ses segments.

Pour une implantation efficace des opérations binaires, il est important de connaître toutes les intersections d'un segment appartenant à un golygone avec tous les segments de l'autre golygone, et ceci dans l'ordre de parcours des frontières. Le dernier champ de la structure *SEGMENT* est nommé *cross1* parce qu'il contient un pointeur vers le premier point d'intersection sur ce segment. Ce dernier, comme on le verra dans la description de la prochaine structure, donne les informations nécessaires pour accéder aux croisements suivants. Un pointeur nul signifie que le segment n'intersecte avec aucun autre.

La structure utilisée pour garder les informations sur un point d'intersection contient, non seulement les coordonnées (x_i, y_i) de ce point, mais aussi les renseignements pertinents permettant d'effectuer les opérations binaires entre deux golygones.

```
#define VISITED 2
typedef struct cross {
        double xi;
        double yi;
        char flag;
        double lambda[2];
        int iseg[2];
        struct cross *nextcr[2];
} CROSS;
```

Comme dans le cas de la structure *COMP*, la variable *flag* est utilisée de façon interne. Présentement, les deux premiers bits servent à réaliser les opérations binaires. Le bit 0 reflète l'indice (0 ou 1) du golygone contenant le segment qui part vers la gauche. Le bit 1 est allumé par la constante *VISITED* lorsque le point d'intersection est visité pour la première fois.

Les variables *lambda*, *iseg* et *nextcr* sont définies comme des tableaux bidimensionnels, le premier et le deuxième élément référant respectivement au premier et au deuxième golygone. Ainsi, lorsque un segment a appartenant au golygone #0 intersecte un segment b appartenant au golygone #1, *lambda*[0] contient la valeur $\lambda_a(b)$ alors que *lambda*[1] contient la valeur $\lambda_b(a)$. L'indice du segment a est placé dans *iseg*[0] et celui du segment b est placé dans *iseg*[1]. Enfin, si le segment a intersecte avec plus d'un segment du golygone #1, nextcr[0] contient un pointeur vers le prochain point d'intersection sur a. Sinon, il contient un pointeur nul. De même en est-il pour le segment b. Tous les points d'intersection sur un même segment sont triés par ordre croissant de λ.

Il est important de noter que, puisqu'une intersection implique deux segments, chaque structure de type *CROSS* est référencée par exactement deux pointeurs. Ainsi, les deux segments impliqués pointent vers la même structure, ce qui permet d'emmagasiner les informations de façon compacte et d'éviter les incohérences.

La figure 3.1 illustre la façon dont sont organisées les données en mémoire, dans la forme la plus générale, c'est-à-dire lorsque deux golygones sont impliqués. Les blocs isolés représentent des éléments détachés alors que des blocs de structures contigües ou reliées par des lignes brisées illustrent une allocation contigüe en mémoire. Les flèches dessinées en lignes pleines et en lignes pointillées représentent respectivement des pointeurs absolus et relatifs. Enfin, une variable annotée par les indices 0 et 1 est un tableau de deux éléments, le premier référençant au golygone A et le second au golygone B. On voit, qu'à partir d'un golygone, il est possible d'accéder à toutes ses composantes et à tous ses segments, et à partir des segments, d'accéder à tous les points d'intersection.

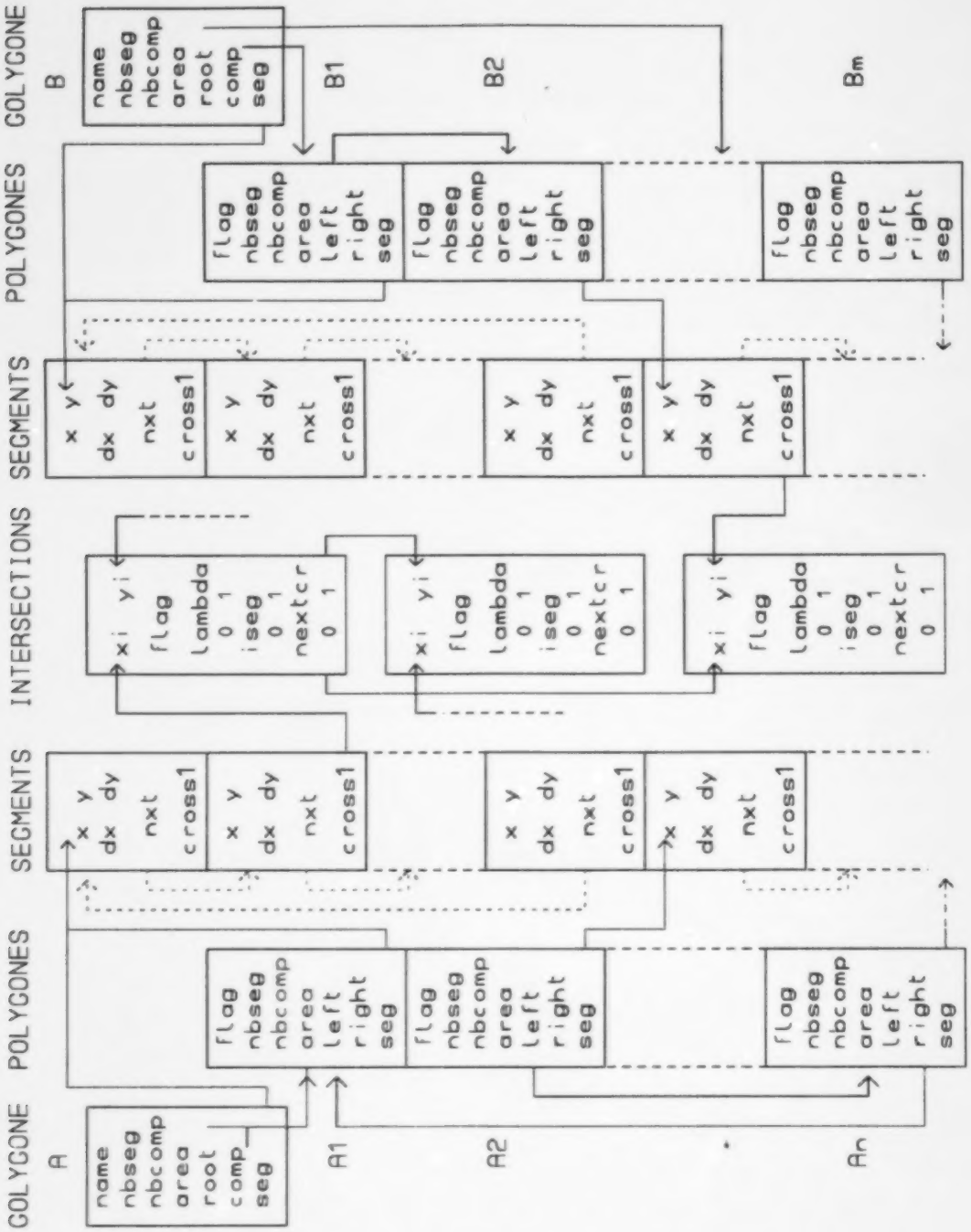


Figure 3.1 Organisation des données en mémoire

III.2 Les procédures

Dans cette section nous décrivons les détails d'implantation des principales procédures. L'implantation a été faite sur le domaine le plus grand possible, c'est-à-dire qu'elle dépasse le cadre théorique de simplicité et normalisation. Toutes les procédures sont réunies sous forme de bibliothèque (*libG.a*) utilisable par le programmeur intéressé à développer des programmes d'application manipulant des golygones. Les noms, descriptions et conditions d'applicabilité des fonctions disponibles sont résumés dans le tableau III et ces fonctions sont présentées en appendice en suivant les conventions du manuel d'utilisation UNIX (UNIX Programmer's Manual, 1983).

Comme nous l'avons déjà précisé, nous avons choisi le langage C comme langage d'implantation. Le fichier *golygon.h* doit être inclus au début de chaque procédure. En plus de la description des structures, il contient les définitions des constantes, des macros et des variables externes. Le rôle de certaines des constantes a déjà été présenté à la section précédente et celui des autres sera spécifié au fur et à mesure des besoins.

```
#define MAXCOMP 1E5
#define MAXSEG 100
#define NAMESIZE 1024
#define DIRINTER 0
#define DIRUNION 1
#define SHARE 1
#define CLOCKWISE 2
#define VISITED 2
#define SAME 4
#define EPSILON 1.0E-6

#define MIN(x, y) (((x) < (y)) ? (x) : (y))
#define MAX(x, y) (((x) >= (y)) ? (x) : (y))
#define MINMAX(y, yl, yh)  (y > yh ? (yh = y) : (y < yl ? (yl = y) : y))

extern int      verbose;
```

nom	description	type	paramètres
cros_seg	vérifie s'il y a intersection entre deux segments	CROSS *	SEGMENT *sa SEGMENT *sb
g_area	calcule la surface d'un golygone	double	GOLYGON *gol
g_binop	effectue une opération binaire entre deux golygones	int	FILE *ofile GOLYGON *gol[2] int dir
g_cmpl	complémente un golygone	int	GOLYGON *gol
g_cross	trouve toutes les intersections entre deux golygones	int	GOLYGON *gol[2]
g_incl	teste l'inclusion d'un point dans un golygone	int	double x, y GOLYGON *gol
g_linear	applique une transformation linéaire à un golygone	int	TRANSMAT *tmat GOLYGON *gol
g_norm	normalise un golygone	int	GOLYGON *gol
g_perim	calcule le périmètre d'un golygone	double	GOLYGON *gol
g_print	écrit un golygone en sortie	int	FILE *ofile GOLYGON *gol
g_read	lit un golygone en entrée	int	FILE ifile GOLYGON *gol
g_simple	teste la simplicité d'un golygone	int	GOLYGON *gol
g_tree	monte l'arborescence d'inclusion	COMP *	GOLYGON *gol
p_area	calcule la surface d'un polygone	double	COMP *pol
p_cmpl	complémente un polygone	int	COMP *pol
p_incl	teste l'inclusion d'un point dans un polygone	int	double x, y COMP *pol
p_perim	calcule le périmètre d'un polygone	double	COMP *pol
p_print	écrit un polygone en sortie	int	FILE *ofile COMP *comp

Tableau III Liste des fonctions accessibles au programmeur

type	nom	signification
double	alpha	angle de rotation
double	area	surface
CROSS*	ca	pointeur vers point d'intersection actif
CROSS*	cl	pointeur vers dernier point d'intersection
CROSS*	cs	pointeur vers une intersection
CROSS*	ct	pointeur temporaire
double	d	déterminant entre deux segments
int	dir	direction à suivre pour réaliser une opération binaire
int	disj	nombre de composantes f-disjointes
double	dxa, dya	valeurs de déplacement en x et y du segment a
double	dxb, dyb	valeurs de déplacement en x et y du segment b
GOLYGON*	gol	pointeur vers un golygone
GOLYGON*	gol[2]	pointeur vers un tableau de golygones
int	i, j, k	compteurs de boucle
FILE*	ifile	fichier d'entrée
int	ip	indice dans le tableau des polygones
int	is	indice du segment courant passif
int	is0	indice dans le tableau des segments du golygone 0
int	is1	indice dans le tableau des segments du golygone 1
int	isa	indice du segment courant actif
double	la	valeur de lambda sur segment a
double	lb	valeur de lambda sur segment b
int	nbc	compteur de composantes
int	nbs	compteur de segments
int	offset	valeur de déplacement vers segment suivant
FILE*	ofile	fichier de sortie
double	perim	périmètre
COMP*	ptc	pointeur vers composante courante
SEGMENT*	pta, ptl	pointeurs vers segment actif et dernier segment
COMP*	ptn	pointeur vers composante suivante
SEGMENT*	pts	pointeur vers segment courant
COMP*	root	pointeur vers racine de l'arborescence d'inclusion
SEGMENT*	sa, sb	pointeurs vers segments a et b
double	x, y	coordonnées du point à tester
double	x1, y1	coordonnées queue d'un segment
double	x2, y2	coordonnées tête d'un segment
double	xh, yh	point supérieur droit
double	xi, yi	coordonnées d'un point d'intersection
double	xl, yl	point inférieur gauche
double	xo, yo	point d'origine des transformations

Tableau IV Liste des principales variables utilisées

Enfin, afin de faciliter la lecture et la compréhension du code, nous avons, le plus possible, utilisé d'une routine à l'autre les mêmes noms pour des variables ayant la même signification. Le tableau IV donne la liste des principales variables utilisées.

III.2.1 Entrée/sortie et validation

Les procédures présentées dans cette section couvrent deux aspects importants. Le premier concerne la compatibilité entrée/sortie, c'est-à-dire le passage du format externe au format interne, et inversement. Le second s'occupe de la réalisabilité des golygones d'entrée, celle-ci n'étant aucunement assurée par le format externe.

Il faut d'abord lire les informations sous format externe et les traduire dans un format interne. C'est la fonction **g_read** qui remplit ce rôle. Elle est chargée d'allouer l'espace mémoire pour le tableau des segments et des composantes, et d'y mettre, selon le cas, les valeurs lues, calculées ou d'initialisation.

Elle commande d'abord l'allocation de l'espace mémoire pour un tableau de *MAXSEG* segments et garde un pointeur (*pts*) vers le premier élément de ce tableau.

```
maxseg = MAXSEG;
gol→seg = (SEGMENT *) malloc(maxseg * sizeof(SEGMENT));
pts = gol→seg;
```

Elle lit ensuite, ligne par ligne, les informations contenues dans le fichier d'entrée. Ceci se fait par une boucle qui s'arrête lorsqu'elle rencontre soit le caractère de fin de golygone, soit celui de fin de fichier. Au fur et à mesure de la lecture, les compteurs de segments (*nbs*) et de composantes (*nbc*) sont incrémentés, et les coordonnées des sommets, de même que les valeurs d'incrément pour aller vers le segment suivant dans une composante, sont emmagasinées dans la structure pointée par *pts*. Si l'espace alloué au début n'est pas suffisant pour le nombre de segments contenus dans le fichier d'entrée, il y a réallocation d'un espace plus grand, de façon à ce que les segments soient toujours placés séquentiellement en mémoire.

```
for (nbs = – 1, ok = 1, nbc = 0, pts = gol→seg – 1; ok;) {
    if (nbs == maxseg – 1) {
    /* le tableau alloué n'est pas assez grand */
        maxseg += MAXSEG;
        gol→seg = (SEGMENT *) realloc(gol→seg, maxseg * sizeof(SEGMENT));
        pts = gol→seg + nbs;
    }
    if (fgets(line, MAXLINE, ifile) == NULL)
    /* le caractère de fin de golygone n'est pas présent */
        line[0] = '$';
    switch (line[0]) {
    case '$':
        if (nbc)
        /* il y a au moins une composante: trouver la valeur d'incrément
        vers le premier segment */
            pts→nxt = offset – nbs;
        ok = 0;
        break;
    case '%':
        if (nbc++)
        /* il y a plus d'une composante */
            pts→nxt = offset – nbs;
        offset = nbs + 1;
        break;
    default:
        nbs++;
        pts++;
        pts→nxt = 1;
        if (sscanf(line, "%lf %lf", &pts→x, &pts→y) != 2) {
            fprintf(stderr, "invalid input: gol:%s ", gol→name);
            fprintf(stderr, "segment:%d comp:%d\ n", nbs – offset, nbc);
            exit(1);
        }
    }
}
```

Lorsque le nombre exact de segments composant le golygone est connu, il y a réallocation de seulement l'espace mémoire nécessaire et, pour chaque segment, les valeurs de déplacement *dx* et *dy* sont calculées et le pointeur *cross1* est mis à *NULL*.

Enfin, il y a allocation d'un tableau de *nbc* composantes et initialisation de tous les champs.

```
gol→comp = (COMP *) malloc(nbc * sizeof(COMP));
for (nbs = 0, ptc = gol→comp, pts = gol→seg; nbs < nbc;
     nbs++, ptc++, pts++) {
          ptc→seg = pts;
          while (pts→nxt == 1)
               pts++;
          ptc→nbseg = 1 - pts→nxt;
          ptc→flag = 0;
          ptc→nbcomp = 0;
          ptc→area = 0;
          ptc→left = NULL;
          ptc→right = NULL;
}
```

La fonction **g_print** joue le rôle inverse de la fonction *g_read*. Elle consiste essentiellement en une boucle qui parcourt le tableau des composantes et, pour chacune d'elle, génère, sur le fichier de sortie spécifié par *ofile*, une ligne contenant le caractère "%". Ensuite, elle appelle la primitive **p_print** en lui passant un pointeur vers la composante courante. Ce pointeur permet à *p_print* de se positionner correctement dans le tableau des segments et d'écrire les coordonnées de tous les segments appartenant au polygone d'entrée. *G_print* génère enfin le caractère de fin de golygone.

La validation d'un golygone consiste à vérifier s'il est réalisable c'est-à-dire, s'il est simple et normalisé. C'est la fonction **g_simple** qui s'occupe de vérifier la simplicité d'un golygone. Au chapitre II, nous avons déjà dit qu'un golygone est simple si aucun de ses segments n'intersecte avec aucun autre. *G_simple* est essentiellement faite de deux boucles imbriquées servant à vérifier, pour toute paire de segments, que ceux-ci n'intersectent pas.

```
for (is0 = 0; is0 < nbs; is0++)
    for (is1 = is0 + ((ps + is0)→nxt == 1 ? 2 : 1); is1 < nbs; is1++)
        if (is1 + (ps + is1)→nxt != is0) {
            s[0] = ps + is0;
            s[1] = ps + is1;
            .....
        }
```

La boucle externe parcourt une fois le tableau des segments alors que la boucle interne se positionne dans le tableau de façon à éviter de tester l'intersection entre deux segments consécutifs dans une même composante. Elle commence donc généralement avec le segment situé deux indices plus loin que celui référencé par la boucle extérieure à moins que celui-ci soit le dernier d'une composante (*nxt* < 1). Dans ce cas, il est nécessaire de vérifier si ce segment intersecte avec le premier segment de la composante suivante. La condition *if* sert à éviter de tester l'intersection entre le premier et le dernier segment d'une même composante.

Cette implantation est celle qui est la plus directe et nous sommes conscients qu'il y a place pour en améliorer l'efficacité. Par exemple, nous pensons à une implantation qui testerait d'abord la simplicité de chacune des composantes et qui vérifierait ensuite la simplicité globale en ne considérant que les composantes qui, prises deux à deux, ne seraient pas exclues par le test dit ''de la boîte''. Celui-ci consiste à trouver les coordonnées la plus petite boîte *logique* qui puisse contenir complètement un polygone. Il est ensuite peu coûteux de vérifier si les domaines définis par deux boîtes sont disjoints et, si c'est la cas, d'éliminer le test d'intersection pour toutes les paires de segments entre deux polygones.

Pour le moment, les tests d'exclusion ne se font qu'au niveau des segments. Pour chaque paire de segments retenus, *g_simple* vérifie s'ils sont raisonnablement près pour que cela vaille la peine de vérifier s'ils intersectent et, le cas échéant, appelle la primitive **cros_seg** en lui passant les pointeurs de segments $s[0]$ (segment a) et $s[1]$ (segment b). C'est *cros_seg* qui calcule les valeurs $\lambda_a(b)$ et $\lambda_b(a)$ selon les formules données à la section II.3.3. Pour que le calcul de ces valeurs ne soit pas inutilement coûteux, il est important de détecter le plus tôt possible les segments qui

n'intersectent pas. Nous avons donc appliqué l'algorithme suivant:

Etape 1: Calculer le dénominateur. S'il est égal à 0, retourner le pointeur nul.

Etape 2: Calculer $\lambda_a(b)$. Si $\lambda_a(b) > 1$ ou $\lambda_a(b) < 0$, retourner le pointeur nul.

Etape 3: Calculer $\lambda_b(a)$. Si $\lambda_b(a) > 1$ ou $\lambda_b(a) < 0$, retourner le pointeur nul.

Etape 4: Si a et b intersectent tous deux en une de leurs extrémités, et qu'au moins une des deux est une tête, retourner le pointeur nul.

Etape 5: Allouer l'espace nécessaire pour le nouveau point d'intersection et y emmagasiner:
- les coordonnées du point d'intersection;
- les valeurs de $\lambda_a(b)$ et $\lambda_b(a)$;
- l'indice du golygone contenant le segment à gauche.

Etape 6: Retourner le pointeur vers la nouvelle structure allouée.

Les étapes 1 et 4 de cet algorithme sont fortement dépendantes du contexte, c'est-à-dire du fait que nous manipulons des golygones et que certains points d'intersection ne sont pas intéressants. Puisque toutes les composantes d'un golygone forment des figures fermées, tout sommet est à la fois tête d'un segment et queue d'un autre. Cette observation nous permet de traiter efficacement le cas des segments parallèles (étape 1) et celui où il y a intersection entre deux sommets (étape 4). Nous décrirons de façon plus détaillée les motivations sous-jacentes à cette implantation dans le section traitant de dégénérescence.

La dernière fonction de ce groupe est **g_norm**. Cette fonction s'occupe d'orienter correctement chacune des composantes d'un golygone en utilisant les informations données par l'arborescence d'inclusion des composantes. Si l'arborescence n'est pas déjà montée, elle le fait en appellant la primitive **g_tree**. Celle-ci trouve d'abord l'aire de chaque composante et, si l'aire est négative, met à jour $ptc \rightarrow flag$.

```
for (ptc = gol→comp, j = 0; j < nbc; j++, ptc++) {
    ptc→area = p_area(ptc);
    if (ptc→area < 0)
        ptc→flag |= CLOCKWISE;
}
```

La valeur absolue de l'aire d'une composante est un renseignement précieux lorsqu'il s'agit de tester, pour deux composantes p et q, l'inclusion naïve de l'une dans l'autre. En effet, pour toute paire de composantes, il n'est nécessaire que de tester l'inclusion de la plus petite dans la plus grande. *G_tree* alloue ensuite l'espace nécessaire pour une composante *bidon* représentant la racine de l'arborescence. Par convention, toutes les composantes du golygone seront naïvement incluses dans cette composante. Enfin, elle monte l'arborescence d'inclusion en y insérant les composantes une à une, selon l'algorithme 1.

Une fois l'arborescence en place, *g_norm* oriente chacune des composantes en fonction de son niveau d'inclusion. Tout polygone englobé par un nombre impair de composantes doit tourner en sens horaire. Tous les autres, doivent tourner dans l'autre sens. La boucle qui se charge de cette opération est codée comme suit:

```
for (ptc = gol→comp, j = 1; j <= nbc; j++, ptc++) {
    odd = ptc→nbcomp & 1;
    /* odd est un booléen qui prend la valeur 1 si 'nbcomp' est impair */

    clockwise = ptc→flag & CLOCKWISE;
    /* clockwise est un booléen qui prend la valeur 1 si */
    /* le deuxième bit de 'flag' est allumé */

    if (odd && !clockwise || !odd && clockwise) {
    /* l'orientation d'une composante doit être changée si, */
    /* soit elle est englobée par un nombre impair de polygones */
    /* et elle est d'orientation antihoraire */
    /* soit elle est englobée par un nombre pair de polygones */
    /* et elle est d'orientation horaire */
        p_cmpl(ptc);
        ptc→flag ^= CLOCKWISE;
    }
}
```

Le golygone obtenu après normalisation est toujours fini. Pour obtenir un golygone infini, il faut faire suivre la normalisation d'une complémentarisation (voir section III.2.3).

III.2.2 Opérations scalaires

Le second groupe de procédures réunit celles qui permettent de réaliser les opérations scalaires telles le calcul de la surface ou du périmètre d'un golygone et le test d'inclusion d'un point dans un golygone.

G_area et **g_perim** consistent tous les deux en une boucle sur tous les éléments du tableau des composantes. Pour chaque composante *ptc*, *g_area* accumule le résultat retourné par l'appel à la primitive *p_area*. De même en est-il de *g_perim* qui appelle *p_perim*.

Afin de n'accéder à chaque segment qu'une seule fois, **p_area** implante l'équation (5) modifiée:

$$\Lambda (p) = \tfrac{1}{2} \sum_{i=0}^{m-1} (x_i \, (y_{i+1} - y_i) - y_i \, (x_{i+1} - x_i)).$$

En effet, puisque les valeurs $x_{i+1} - x_i$ et $y_{i+1} - y_1$ sont déjà disponibles dans la structure *SEGMENT*, il n'est pas nécessaire, pour obtenir le résultat cherché, d'avoir deux pointeurs vers les deux segments impliqués.

```
for (a = 0, s = ptc→seg, is = 0; is < ptc→nbseg; is++, s++)
    a += s→x * s→dy – s→y * s→dx;
return (a * 0.5);
```

De la même façon, **p_perim** utilise directement les valeurs *dx* et *dy* pour calculer la longueur des segments.

G_incl est la procédure qui sert à vérifier si un point x, y est inclus ou non dans un golygone g. Pour cela, elle appelle la primitive *p_incl*, pour chacun des polygones composant g, et fait la somme des valeurs retournées. Comme nous l'avons déjà dit au chapitre précédent, la signification du résultat obtenu dépend du fait que le golygone soit normalisé et qu'il soit fini ou non.

P_incl sert à tester l'inclusion naïve d'un point dans un polygone en utilisant la méthode décrite à la section II.3.2. Il s'agit donc de vérifier, pour chacun des segments, s'il intersecte la droite horizontale passant par le point à tester et, le cas

échéant, si ce point d'intersection est situé du bon côté; dans notre cas, nous avons choisi le côté gauche. Ensuite, le compteur correspondant à la direction du segment impliqué (*pos*, pour un segment allant de haut en bas et *neg*, pour un segment allant de bas en haut) est incrémenté. Lorsque tous les segments ont été visités, *p_incl* retourne la valeur (*pos* − *neg*).

Afin que cette procédure ne soit pas trop coûteuse, on ne calcule le point d'intersection d'un segment avec la droite horizontale passant par le point à tester, que lorsque c'est vraiment nécessaire. Ainsi, un segment dont les deux extrémités sont, soit au-dessus, soit au-dessous de la droite, est immédiatement rejeté. De même en est-il d'un segment dont les deux extrémités sont à droite du point.

Reste à considérer le cas dégénéré où la droite horizontale passe par un des sommets du polygone. Ce cas est beaucoup traité dans la littérature: citons, à titre d'exemples, Preparata et Shamos (1985) et, plus récemment, Knott (1987). Il s'agit de déterminer si un point d'intersection coïncidant avec un sommet compte pour un ou deux points d'intersection. Si les deux segments impliqués sont situés de chaque côté de la droite (figure 3.2-A,B), il n'y a logiquement qu'un seul point d'intersection puisqu'en déplaçant légèrement vers le haut (ou vers le bas) le point à tester, la droite ne passe plus par ce sommet et il n'y a plus qu'un des deux segments qui intersecte cette droite. Par contre, si les segments sont tous les deux du même côté, il y a, à la limite, soit deux intersections (figure 3.2-C,D), soit aucune (figure 3.2-E).

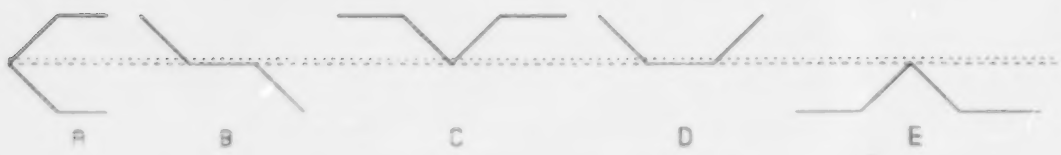


Figure 3.2 Cas dégénérés d'inclusion

Une solution élégante à ce problème est de ne considérer une intersection avec une extrémité d'un segment, que si la valeur en ordonnée de cette extrémité est inférieure à la valeur en ordonnée de l'autre extrémité du segment (Preparata et Shamos, 1985). Ainsi, lorsqu'un point d'intersection est retenu, nous ne tenons compte que des segments négatifs qui n'intersectent pas à leur tête, et des segments positifs qui n'intersectent pas à leur queue, ce qui est codé comme suit:

```
if ((xi = x1 + (y – y1) * pts→dx / pts→dy) <= x)
    if (y2 > y1 && y2 != y)
        neg++;
    else if (y2 < y1 && y1 != y)
        pos++;
```

III.2.3 Opérations unaires

Le troisième groupe de procédures réunit celles qui permettent de réaliser les opérations unaires de complémentarisation et de transformations linéaires.

Comme nous l'avons déjà dit à la section II.4.1, la complémentarisation d'un golygone se fait en complémentant chacune de ses composantes. **G_cmpl** est une routine similaire à *g_area* et *g_perim*. Elle s'occupe d'appeler, pour chacun des polygones composant le golygone, la procédure qui fera le véritable travail de complémentarisation. Cette procédure, appelée **p_cmpl**, fait l'échange *in situ* de paires de segments. Pour cela, nous avons besoin de deux pointeurs, indiquant les bornes du tableau de segments à interchanger, et de deux variables intermédiaires capables de garder les valeurs des coordonnées d'un des deux segments impliqués dans l'échange.

Au départ, le premier pointeur réfère au premier segment du polygone alors que le second réfère au dernier. Lorsque les coordonnées de ces deux segments sont échangées, le premier pointeur avance vers le prochain segment et le second recule vers le précédent, diminuant de deux segments le tableau qui reste à complémenter.

```
nbs = comp→nbseg;
/* trouver pointeur vers le premier segment */
pt1 = comp→seg;
/* trouver pointeur vers le dernier segment */
pt2 = pt1 + nbs - 1;
/* plancher de la division par 2 pour connaître le nombre d'itérations */
nbs >>= 1;
/* échanger les coordonnées des segments 2 à 2 */
for (k = 0; k < nbs; k++) {
    x1 = pt1→x;
    y1 = pt1→y;
    pt1→x = pt2→x;
    pt1→y = pt2→y;
    pt2→x = x1;
    pt2→y = y1;
    pt1++;
    pt2--;
}
```

Il ne reste ensuite qu'à recalculer toutes les valeurs de déplacement en x et y.

```
for (k = 0, pt1 = comp→seg; k < comp→nbseg; k++, pt1++) {
    pt2 = pt1 + pt1→nxt;
    pt1→dx = pt2→x - pt1→x;
    pt1→dy = pt2→y - pt1→y;
}
```

Le deuxième type d'opération unaire est la transformation linéaire. C'est la fonction **g_linear** qui réalise cette opération en appliquant, aux coordonnées de chacun des sommets du golygone, la matrice de transformation reçue en entrée. Cette matrice de transformation est implantée comme une structure à six (6) champs correspondants aux six valeurs de transformation présentées à la section II.4.2.

```
typedef struct transmat {
    double  a, b, c, d, e, f;
} TRANSMAT;
```

Quoique les valeurs associées à ces champs puissent être directement fournies par le programmeur, elles peuvent aussi être obtenues par applications successives de

procédures servant à modifier cette matrice. Ainsi, les fonctions **t_matrix**, **s_matrix** et **r_matrix** servent respectivement à ajouter, à la matrice de transformation courante, une translation, un changement d'échelle et une rotation, alors que **x_matrix** et **y_matrix** réalisent les désaxements en x et y. Toutes ces fonctions reçoivent comme premier paramètre, un pointeur vers la matrice de transformation courante et comme paramètres suivants, les valeurs de transformation. Enfin, sauf pour *t_matrix*, les coordonnées du point d'origine de la transformation demandée sont spécifiées comme derniers arguments.

Les calculs effectués dans chacune de ces procédures reflètent exactement les calculs matriciels exposés à la section II.4.2. A titre d'exemple, *s_matrix* est codé comme suit:

```
s_matrix(tmat, sx, sy, xo, yo)
    TRANSMAT *tmat;
    double sx, sy;
    double xo, yo;

{
    tmat→a *= sx;
    tmat→b *= sy;
    tmat→c *= sx;
    tmat→d *= sy;
    tmat→e = tmat→e * sx – sx * xo + xo;
    tmat→f = tmat→f * sy – sy * yo + yo;
}
```

III.2.4 Opérations binaires

Le dernier groupe de procédures sert à réaliser les opérations binaires entre deux golygones. Pour le moment, les opérations implantées sont l'union et l'intersection.

La procédure qui est responsable de l'appel, direct ou indirect, de toutes les autres procédures impliquées dans la réalisation d'une opération binaire entre deux golygones s'appelle **g_binop**. Elle reçoit, comme paramètres, un tableau de dimension deux dont les éléments sont de type *GOLYGON*, et la variable *dir* contenant la valeur 0 lorsque l'opération désirée est l'intersection et 1 lorsque c'est l'union. Nous verrons la raison de ces valeurs un peu plus loin.

Avant de faire appel à toute autre routine, *g_binop* s'occupe de vérifier si un des deux golygones correspond à l'ensemble vide ou à l'ensemble universel. En effet, dans ce cas, le résultat est trivial mais une vérification est nécessaire si nous voulons que le principe du *filtre* soit applicable en toute généralité, c'est-à-dire qu'il soit possible d'utiliser le résultat d'une opération binaire comme entrée pour une autre opération binaire, quel que soit ce résultat. Par exemple, nous voudrions pouvoir exécuter $(p \cap q) \cap t$, $p, q, t \in \mathbf{P}$, même si $p \cap q = \varnothing$. La représentation interne de ces deux ensembles ne cause aucun problème. Puisque l'ensemble vide correspond à un golygone ne contenant aucune composante, on affecte un pointeur nul à *gol→comp*, alors que pour l'ensemble universel, c'est à la variable *gol→comp→seg* qu'il faut affecter le pointeur nul. Lorsqu'un golygone vide est détecté, le résultat de l'union est immédiatement obtenu en donnant en sortie (fonction *g_print*) l'autre golygone d'entrée. Par contre si l'opération demandée est l'intersection, le résultat est égal à $\varnothing$. Dans le cas d'un golygone égal à $\mathbf{R}^2$, il faut appliquer le raisonnement inverse.

Une fois ces cas dégénérés traités, il y a appel à la fonction **g_cross** qui cherche tous les points d'intersection entre les deux golygones, marque les composantes impliquées dans ces intersections

ptc→flag |= SHARE

et retourne à *g_binop* le nombre de composantes f-disjointes.

Pour vérifier si deux segments intersectent, *g_cross* fait appel à la procédure *cros_seg* déjà présentée à la section III.2.1. Lorsqu'un point d'intersection est trouvé, une nouvelle structure *CROSS* est allouée et les valeurs calculées y sont emmagasinées. Entre autres, l'indice (0 ou 1) du golygone contenant le segment qui part vers la gauche est utilisé pour initialiser le bit le moins significatif de la variable *flag*. Il correspond au signe du déterminant formé par les deux vecteurs (segments). Ainsi, pour aller à gauche (intersection), il faut continuer sur le segment appartenant à *gol*[0] si le déterminant est positif, et à *gol*[1] s'il est négatif. Notons également que, le nombre de points d'intersection pouvant être assez élevé, il y a vérification de la disponibilité de l'espace mémoire.

```
cs = (CROSS *) malloc (sizeof(CROSS ));
if (cs == NULL) {
    fprintf(stderr, "Cros_seg: too many crossing points. ");
    fprintf(stderr, "No more space available\ n");
    exit(-1);
}
cs→xi = sa→x + la * dxa;
cs→yi = sa→y + la * dya;
cs→lambda[0] = la;
cs→lambda[1] = lb;
cs→flag = (d < 0);
```

Cette structure est référencée par les deux segments participant à l'intersection. Tous les points d'intersection sur un même segment sont triés par ordre croissant de lambda et la variable *cross1* associée à ce segment pointe vers le point d'intersection le plus près de la queue c'est-à-dire celui de plus petit lambda. Chaque point d'intersection pointe à son tour, par l'intermédiaire des pointeurs *nextcr*[0] et *nextcr*[1], vers le prochain point d'intersection sur chacun des deux segments impliqués. Ainsi, à partir de *cross1*, il est possible de suivre, dans l'ordre de parcours du golygone, tous les croisements.

```
for (ip = 0; ip < 2; ip++) {
    ct = pts[ip]→cross1;
    cl = NULL;
    while (ct != NULL && (dl = (ct→lambda[ip] – cs→lambda[ip]) <= 0)) {
    /* deuxième critère d'ordonnancement si deux lambdas sont égaux */
        if (!dl && ct→lambda[1 – ip] > cs→lambda[1 – ip])
            break;
        cl = ct;
        ct = ct→nextcr[ip];
    }
    if (cl == NULL)
        pts[ip]→cross1 = cs;
    else
        cl→nextcr[ip] = cs;
    cs→nextcr[ip] = ct;
}
```

S'il existe au moins une composante f-disjointe et si l'arborescence d'inclusion des deux golygones n'est pas montée, *g_binop* s'occupe de bâtir ces arborescences en appellant la fonction **g_tree** dans laquelle est directement implanté l'algorithme 1. L'aire de chacune des composantes est d'abord trouvée et la variable *flag* est marquée pour refléter l'orientation de chaque polygone.

```
if (ptc→area < 0)
    ptc→flag |= CLOCKWISE;
```

Les composantes sont ensuite insérées une à une dans l'arborescence courante en la parcourant en pré-ordre. Puisque les polygones formant un golygone ne peuvent intersecter entre eux, il suffit de tester l'inclusion d'un point d'un polygone p dans un polygone q pour connaître la relation de descendance entre p et q.

```
if (fabs(q→area) > fabs(p→area)) {
    x = p→seg→x;
    y = p→seg→y;
    if (k = (p_incl(x, y, q))) {
      q = q1→left;
    else
      q = q1→right; } }
```

Toutes les informations nécessaires pour obtenir le résultat de l'opération binaire demandée sont maintenant disponibles et *g_binop* peut faire appel à la routine **dobinop** dans laquelle est implanté l'algorithme 5.

Premièrement, il faut considérer toutes les composantes qui partagent au moins un point d'intersection avec l'autre golygone. Nous avons déjà dit, au chapitre II, que les points d'intersection entre deux golygones sur lesquels on veut effectuer une opération binaire sont les seuls points où il faille prendre une décision de continuer ou non sur le golygone courant. L'indice du golygone à choisir est déterminé par une combinaison des informations données par les variables *cross→flag* et *dir*. Par exemple, on sait que pour obtenir l'union de deux golygones, il faut toujours suivre les frontières les plus extérieures. Par conséquent, il faut choisir le golygone auquel appartient le segment qui traverse l'autre golygone de gauche à droite. Sachant que *cross→flag* & 1 donne l'indice du golygone contenant le segment qui coupe de droite à gauche, et en posant *dir* = 1 pour l'union, l'indice du golygone contenant le segment qui coupe de gauche à droite est obtenu par l'expression suivante:

igol = (cross→flag & 1) ^ dir;

De même, en posant *dir* = 0, cette expression nous donne l'indice du golygone à suivre pour obtenir l'intersection.

Toute nouvelle composante du golygone résultat est initiée par un point d'intersection qui n'a pas encore été visité. Un tel point d'intersection marque le début d'un cycle actif c'est-à-dire d'un parcours des frontières des golygones d'entrée pendant lequel tous les sommets et les points d'intersection rencontrés font partie du résultat. Lorsqu'un point d'intersection est visité pour la première fois, il est étiqueté afin qu'il ne puisse participer à plus d'un cycle actif.

```
for (sp = gol[0]→seg, is = 0; is < gol[0]→nbseg; is++, sp++)
    for (cr = sp→cross1; cr != NULL; cr = cr→nextcr[0])
        if (!(cr→flag & VISITED)) {
            /* début d'un cycle actif */
            fprintf(ofile, "%%\n");
            comp++;
            ca = cr;
            do {
                if (ca→flag & VISITED)
                    break;
                fprintf(ofile, "%f %f\n", ca→xi, ca→yi);
                ca→flag |= VISITED; /* marquer la visite */
                igol = (ca→flag & 1) ^ dir;
                isa = ca→iseg[igol];
                sa = gol[igol]→seg + isa;
                ca = ca→nextcr[igol];
                while (ca == NULL) {
                    isa += sa→nxt;
                    sa = gol[igol]→seg + isa;
                    fprintf(ofile, "%f %f\n", sa→x, sa→y);
                    ca = sa→cross1;
                }
            } while (ca != cr);
        }
```

S'il n'y avait aucune composante f-disjointe, le résultat est complet dès que tous les points d'intersection ont été visités. Dans le cas contraire, il reste à choisir les composantes f-disjointes qui participent au résultat. Afin de savoir si une composante f-disjointe appartenant au golygone i, est incluse dans l'autre golygone $(1 - i)$, nous avons choisi de ne pas utiliser la fonction *g_incl* mais de plutôt faire appel à la fonction **incltree**. Celle-ci s'occupe de trouver le niveau d'inclusion d'une composante (*ptc→nbcomp*) en se servant des renseignements fournis par l'arborescence d'inclusion, ce qui est beaucoup moins coûteux que de parcourir toutes les composantes, comme le fait la fonction *g_incl*. En combinant cette information à la connaissance du signe du golygone il est possible de savoir si la composante est incluse ou non.

```
for (i = 0; i < 2; i++)
    for (j = 0, ptc = gol[i]→comp; j < gol[i]→nbcomp; j++, ptc++) {
        if (!(ptc→flag & SHARE) && ptc→area != 0 && !(ptc→flag & SAME)) {
        /* f-disjointe, non dégénérée et non identique à une autre */
            ptc→nbcomp = 0;
            incltree(gol[1 - i]→root, ptc);
            odd = ptc→nbcomp & 1;
            fini = (gol[1 - i]→area > 0 ? 1 : 0);
            if (odd && fini || !odd && !fini) {
                /* ptc est incluse dans l'autre golygone */
                if (!dir) {
                    fprintf(ofile, "%%\ n");
                    comp++;
                    p_print(ofile, ptc);
                }
            } else {
                if (dir) {
                    fprintf(ofile, "%%\ n");
                    comp++;
                    p_print(ofile, ptc);
                }
            }
        }
    }
```

Enfin, *g_binop* libère l'espace alloué pour tous les points d'intersection et retourne le nombre de composantes dans le golygone résultat.

III.2.5 Problèmes de dégénérescence

Lors de l'implantation de certaines routines, nous avons fait face à des problèmes causés par deux types de cas dégénérés. Le premier concerne la dégénérescence des golygones eux-mêmes, ou d'une de leurs composantes, et le second concerne la relation spatiale qui existe entre deux éléments, telles les composantes dont les segments se chevauchent et celles se touchant en leurs sommets.

Une composante d'entrée est dégénérée si elle comporte moins de trois sommets distincts. L'ensemble vide et l'ensemble $\mathbf{R}^2$ sont deux exemples particuliers de telles composantes. Pour la plupart des opérations, ces deux ensembles ne présentent pas beaucoup d'intérêt, à l'exception des opérations binaires dont nous avons déjà discuté lors de la présentation de la primitive *g_binop*. Pour ce qui est des polygones réduits à un point, ou à un segment, nous pouvons leur appliquer sans problèmes des transformations géométriques et des opérations scalaires. Par contre, lorsque ces composantes sont impliquées dans des opérations binaires, elles ne constituent pas une entrée significative car leur frontière ne peut pas induire de changement d'état d'appartenance. En effet, les points à l'intérieur d'une composante ont normalement un état d'appartenance différent des points à l'extérieur. On ne peut cependant pas parler d'intérieur d'un polygone lorsque ce dernier est dégénéré. On ne peut pas plus parler d'orientation puisque le surface est nulle et que c'est le signe de la surface qui reflète l'orientation. La solution la plus appropriée est d'éliminer ces composantes dès qu'elles sont détectées, ce qui est simple puisque leur surface est toujours nulle.

Le deuxième type de dégénérescence implique deux éléments. Par exemple, un élément peut être un point et l'autre un polygone. Quoiqu'il soit facile de déceler si un point (x, y) appartient à la frontière d'un polygone, il n'est cependant pas évident de décider de son inclusion dans ce polygone.

A titre d'exemple, considérons un polygone p et son complément $\overline{p}$. Les points frontières sont exactement les mêmes. Alors, si on teste l'inclusion d'un de ces points dans p, on aimerait que la réponse soit différente de celle obtenue en testant l'inclusion du même point dans $\overline{p}$. En effet, un polygone est un cas particulier de golygone et, par la définition donnée à la section II.4.1, tout point inclus dans un

golygone est exclu de son complément. Pourtant, la fonction *p_incl* doit retourner la même réponse dans les deux cas puisqu'il y a le même nombre d'intersections entre les segments des polygones et la droite horizontale passant par ce point. Nous avons pris la décision, lorsqu'un tel cas se présente, de considérer un point frontière inclus dans un polygone positif et exclu d'un polygone négatif.

Les deux éléments peuvent aussi être deux polygones. Faire l'union, ou l'intersection, de deux polygones peut présenter des difficultés lorsque ceux-ci se touchent en un seul point ou lorsque deux de leurs segments se chevauchent. La figure 3.3 illustre douze de ces cas (noter que A représente deux polygones identiques et que B en représente un dans un autre). Nous pouvons remarquer qu'ils présentent tous un point commun: si une des deux composantes est légèrement déplacée dans une certaine direction, il n'y a plus de problèmes parce que, ou les deux composantes deviennent complètement séparées, ou alors elles intersectent au moins en deux points.

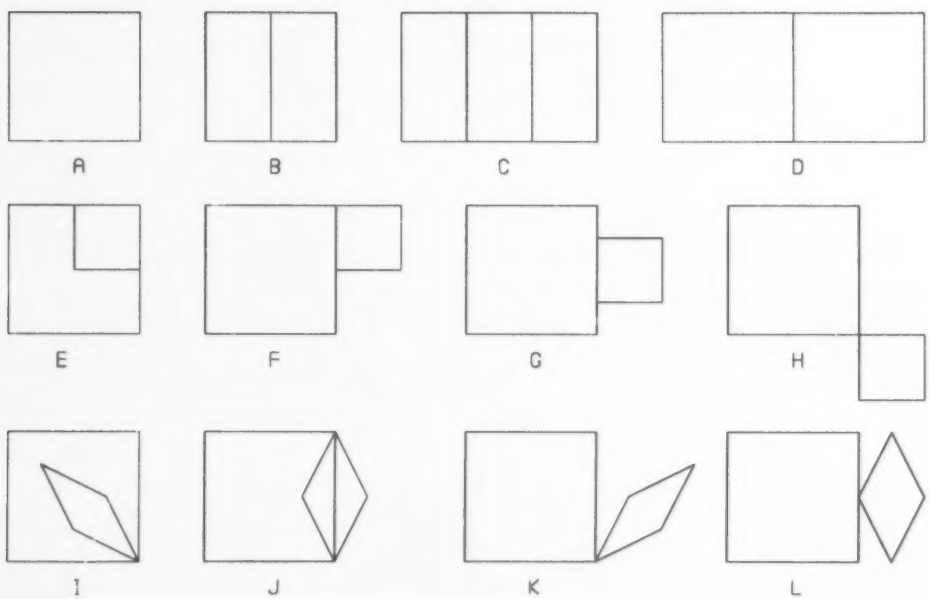


Figure 3.3 Divers cas de dégénérescence

L'approche de "perturbation" présente donc un attrait théorique certain pour résoudre ce type de problèmes. Cependant, cette solution n'est pas élégante puisque la même procédure ne peut être utilisée en toute généralité. Les cas problèmes doivent d'abord être décelés puis traités de façon spéciale. Voyons comment l'implantation de la primitive *cros_seg*, telle que décrite à la section III.2.1, peut s'avèrer un moyen efficace de résoudre la plupart des cas de dégénérescence entre deux polygones, sans perturbation des éléments d'entrée.

Parallélisme

Soit a_i et b_j deux segments parallèles, $a_i \in gol[0]$, $b_j \in gol[1]$. Il est impossible d'appliquer les formules classiques pour trouver les valeurs des λ respectifs puisque la valeur du dénominateur est égale à 0. En fait, dans ce cas, a_i et b_j peuvent ne pas intersecter du tout mais ils peuvent aussi se chevaucher et alors il y a une multitude de points d'intersection. Cependant, lorsque nous désirons faire l'union de deux golygones il n'est pas important de déterminer s'il y a intersection entre deux segments parallèles. En effet, tant que les segments se chevauchent, on peut tout aussi bien être sur l'un que sur l'autre. Le point d'intersection intéressant est celui où la décision de changer ou non de golygone doit se prendre et cela ne peut se faire que lorsque les segments ne sont plus parallèles. Trois cas sont possibles:

1) a_i et b_j vont dans le même sens (figure 3.4 A),
2) les têtes de a_i et b_j font partie du chevauchement (figure 3.4 B),
3) les queues de a_i et b_j font partie du chevauchement (figure 3.4 C).

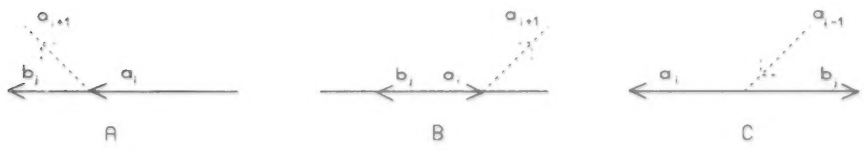


Figure 3.4 Trois combinaisons entre deux segments parallèles

Dans le premier cas, supposons, sans perte de généralité, que $\hat{a}_i$ soit sur b_j. Alors il y a intersection entre $\dot{a}_{i+1}$ et b_j et c'est à ce point d'intersection qu'il y aura décision d'emprunter a_{i+1} ou de continuer sur b_j. Dans le deuxième cas, si on est sur $gol[1]$, $\hat{a}_i$ est rencontrée avant $\hat{b}_j$ et constitue un point de décision qui, comme précédemment, correspond aussi à l'intersection entre b_j et a_{i+1}. On peut faire le raisonnement symétrique lorsqu'au départ on est sur $gol[0]$. Le troisième cas est un peu différent. Alors qu'avec les deux premiers le point de décision était toujours à la tête d'un des deux segments parallèles et, par conséquent, à la queue du segment suivant, nous avons maintenant le cas inverse. Lorsqu'on est sur $gol[1]$, le premier point intéressant rencontré est $\dot{a}_i$, ou de façon équivalente, $\hat{a}_{i-1}$. Par conséquent, le point d'intersection sera trouvé, non plus avec le segment suivant, mais avec le précédent.

On voit donc que, lorsque *cros_seg* décèle un cas de parallélisme, le pointeur nul est immédiatement retourné puisque tout point d'intersection intéressant implique toujours au moins deux segments qui ne sont pas parallèles.

Intersection entre deux sommets

Supposons deux composantes se touchant en un de leurs sommets. Soit a_i, b_j les segments dont la tête est au sommet v_i, et soit a_{i+1}, b_{j+1} ceux ayant v_i comme queue. Alors, si on considère qu'il y a intersection pour $0 \leq \lambda \leq 1$, il y aura quatre points d'intersection au point v_i. Par contre, un seul de ces points suffit pour décider du segment à emprunter. Puisque le parcours des golygones se fait dans le sens de leurs segments, il devrait être suffisant de ne considérer, dans ce cas, que l'intersection entre a_{i+1} et b_{j+1} c'est-à-dire celle impliquant les deux queues. Quoique prometteuse, cette approche ne fonctionne cependant pas dans tous les cas. Voyons, en effet, ce qui se passe lorsqu'on applique l'algorithme 5.

Il faut d'abord se positionner au premier (et seul) point d'intersection: celui-ci fait partie du résultat. On suit alors les segments, à gauche ou à droite selon l'opération désirée, en mettant chaque sommet dans le résultat et jusqu'à rencontrer un point d'intersection déjà visité. Il y a donc nécessairement formation d'un cycle.

Si les deux composantes sont logiquement imbriquées, comme dans l'exemple I de la figure 3.3, le résultat obtenu sera bien celui escompté. Par contre, si elles sont logiquement séparées, comme dans l'exemple K, il y a toujours formation d'un cycle mais celui-ci ne peut être une solution valide puisqu'intuitivement, on sait que l'union devrait être faite des deux composantes, alors que l'intersection devrait être réduite au seul point d'intersection. Pour que l'algorithme fonctionne il faut, soit ignorer le point d'intersection, soit le dédoubler. Dans la première éventualité, les deux composantes sont traitées comme si elles étaient séparées, alors que dans la seconde, il devient possible de tourner autour du point dans le cas de l'intersection ou faire le tour des deux composantes dans le cas de l'union.

Puisque dans un cas, un seul point d'intersection suffit, mais dans un autre il en faut au moins deux, nous avons analysé différentes combinaisons afin d'essayer de déterminer quels points il fallait garder pour avoir toujours la bonne solution lorsque deux polygones intersectent sur un sommet. Nous avons testé chacune des sept approches suivantes sur chacun des exemples de la figure 3.3:

1. ne garder que l'intersection impliquant deux queues;
2. ne garder que les deux intersections tête-queue;
3. ne garder que les deux intersections tête-tête ou queue-queue;
4. garder trois intersections - ignorer une intersection tête-queue;
5. garder trois intersections - ignorer l'intersection tête-tête;
6. garder les quatre intersections;
7. ignorer les quatre intersections.

Le tableau V résume les résultats obtenus, tant pour l'union que pour l'intersection. Les entrées marquées d'un "X" sont celles où le résultat est faux alors que celles marquées d'un "+" représentent un résultat correct mais où certains sommets sont répétes trois fois ou plus. Notons, cependant, que des points répétés ne causent aucun problème.

Nous pouvons observer que, malheureusement, aucune des approches n'offre de solution parfaite pour toutes les situations, et que des cas en apparence très sem-

blables ne peuvent être traités de la même façon. La première approche est cependant celle qui est la plus satisfaisante puisqu'elle peut résoudre tous les cas sauf ceux où deux composantes, logiquement séparées, se touchent en un sommet et où les segments impliqués dans l'intersection ne sont pas parallèles. Entre autres, elle permet de déceler les cas où deux composantes sont identiques: en effet, avec cette implantation, il n'y a aucune intersection entre deux composantes identiques. Il suffit donc de tester, lorsque deux composantes n'intersectent pas et ont des surfaces égales, si un quelconque sommet de l'une coïncide avec un sommet de l'autre. En pareil cas, les composantes sont identiques.

	Approche						
Cas	1	2	3	4	5	6	7
A		X	X		X	X	X
B		X			X	X	
C							X
D			+	+		+	
E		X			X	X	
F		X	+	+		+	
G							
H				X			
I		X		+	+	+	X
J		X	+	X	X	X	X
K	X	X	X		X	X	
L							

Tableau V Résultats pour diverses approches de traitement de cas dégénérés

Nous avons donc décidé de garder celle-ci et, pour le moment, de simplement émettre un message d'avertissement lorsque le résultat peut ne pas être correct. Ceci

permet à l'utilisateur de vérifier lui-même le résultat obtenu et de le corriger, s'il y a lieu, soit en appelant une procédure traitant spécifiquement ce type de dégénérescence, soit en modifiant légèrement les coordonnées du sommet impliqué. D'ailleurs cette solution peut parfois être la seule acceptable lorsque, comme dans le cas K, le résultat attendu de l'union donnerait, sans perturbation, un golygone qui n'est pas simple.

III.3 Les programmes

Dans cette section nous décrivons l'utilisation des différentes commandes disponibles et donnons quelques exemples d'appels et de résultats.

Tous nos programmes ont été compilés et roulent avec succès sur SUN 3/60, UNIX PC AT&T et Symmetric 375. Une version sur système d'exploitation DOS existe aussi, compilée avec Turbo C. Le développement s'est fait dans un environnement UNIX et nos programmes suivent la philosophie de ce système d'exploitation. Entre autres, en ce qui a trait au format des lignes de commande, nous avons opté pour le standard " *UNIX System V* " (Thomas et coll., 1986). Ainsi, le format général de nos commandes est:

programme [option [s]] [argument [s]] fichier [s] .

Le *programme* est le nom de la commande à exécuter. Les *options* servent, comme leur nom l'indique, à faire des demandes facultatives au programme. Elles peuvent être de deux types: les options sans argument, et celles suivies d'un nombre prédéfini d'*arguments*. Ces arguments servent à définir de façon précise l'action requise par l'option. Plusieurs options peuvent être regroupées mais il ne peut y avoir qu'une seule option avec argument par groupe. Un groupe d'options commence toujours par le caractère "–". Certaines commandes requièrent un argument supplémentaire entre les options et les fichiers. Ce paramètre obligatoire est utilisé pour préciser à la commande le type exact d'opérations désirées. Il est formé par une ou plusieurs lettres (sans séparateur), chacune correspondant à une opération spécifique. A titre d'exemple, soit la ligne de commande suivante:

unop – vo 10 10 – r 3.1416 lc fichier ;

comme nous le verrons un peu plus loin, cette commande sert à effectuer des opérations unaires; le paramètre *lc* spécifie le type exact d'opérations unaires, en l'occurence la transformation linéaire et la complémentarisation. Un premier groupe d'options (*–vo*) est composé d'une option sans argument (*v*) et d'une autre devant être suivie de deux arguments (*o* arg1 arg2). Un second groupe ne contient que l'option *r* qui s'attend à recevoir un seul argument.

Les *fichiers* servent à dire au programme où trouver les données d'entrée. Si le nom d'un fichier est remplacé par un "–", c'est le fichier d'entrée standard (*stdin*) qui est utilisé. Les résultats sont toujours envoyés sur le fichier de sortie standard (*stdout*), et tous les messages sur le fichier d'erreur standard (*stderr*). Les messages sont réduits au minimum, à moins d'utiliser l'option "verbose" (*–v*). Tout ceci permet d'employer nos commandes comme des filtres, c'est-à-dire que la sortie de l'une peut directement servir d'entrée à une autre.

Notre logiciel se compose actuellement de trois programmes principaux (commandes **scalop**, **unop** et **binop**) et de quatre utilitaires (commandes **gcut**, **gcat**, **ggraf** et **gmake**). Les programmes principaux correspondant essentiellement aux trois grandes classes d'opérations, ils nécessitent tous au moins un paramètre servant à préciser, parmi toutes les opérations de cette classe, celles qui doivent effectivement être exécutées. Comme pour les procédures, les pages du manuel d'utilisation pour chacune de ces commandes sont fournies en appendice. Elles décrivent en détails le rôle de chaque option et argument. Afin de faciliter la mémorisation de la syntaxe et de ne pas obliger l'utilisateur à avoir constamment recours à cette documentation, nous avons essayé de choisir des lettres qui soient significatives, tant en anglais qu'en français, des options et des opérations disponibles. Dans les pages de manuel, nous précisons les mots anglais qui définissent chacune des options.

La commande *scalop* sert à effectuer des opérations scalaires sur un ou plusieurs golygones d'entrée. En plus des opérations décrites aux sections II.3 et III.2.2, elle permet aussi d'obtenir le nombre de polygones composant chacun des golygones d'entrée, de même que le nombre de segments. Plusieurs opérations peuvent être demandées lors du même appel de cette commande. Lorsque l'option *–v* est utilisée, le format de sortie obtenu est très explicite. Par exemple, la commande

```
scalop – v  acnp  lacs
```

donne, comme résultat:

g-polygon LACS
 # components : 35
 # segments : 787
area : 2419.275442
perimeter : 1604.798178

Cependant, par défaut, tous les résultats concernant un golygone sont rassemblés sur une même ligne, ce qui permet d'utiliser la sortie obtenue avec d'autres commandes UNIX, telle *awk*. Ainsi, si l'on désire avoir les mêmes informations que dans l'exemple précédent mais, cette fois-ci, pour les golygones des figures 1.1 (fichier *suisse*), 1.2 (fichier *lacs*) et 1.3 (fichier *prot*), la commande

scalop apcn suisse lacs prot

donne:

Suisse	**1**	**440**	**41116.35**	**1612.81**
Lacs	**35**	**787**	**2419.28**	**1604.80**
Protestants	**21**	**937**	**21244.67**	**3050.10**

On peut remarquer que l'ordre dans lequel les informations sont données étant fixé par le programme, il ne correspond pas nécessairement à celui sur la ligne de commande.

La commande *unop* permet d'effectuer les opérations unaires de normalisation et de complémentarisation ainsi que les transformations linéaires. Si plusieurs opérations sont demandées lors du même appel, elles seront toujours exécutées dans l'ordre 1] transformation, 2] normalisation, 3] complémentarisation, et cela quel que soit l'ordre dans lequel sont agencées les lettres formant le paramètre spécifiant ces opérations. Par contre, toutes les transformations linéaires sont appliquées dans l'ordre d'apparition des options sur la ligne de commande, ce qui permet à l'utilisateur de créer une matrice de transformation aussi compliquée qu'il le veut et d'obtenir ainsi n'importe quelle transformation linéaire avec un seul appel à la commande *unop*. Ainsi les deux commandes suivantes, bien qu'ayant exactement les mêmes options, ne donneront pas le même résultat:

```
unop -o 20 20 -s 1.5 1.5 -t 5 -5 -o 0 0 -r 2 1 fig2.1
unop -t 5 -5 -o 20 20 -r 2 -s 1.5 1.5 -o 0 0 1 fig2.1
```

Dans le premier cas, le changement d'echelle utilise comme point d'ancrage les coordonnées spécifiées par l'option *–o*; la translation est ensuite exécutée. Enfin, le point d'origine courant est remis à la valeur de défaut (0, 0) avant qu'une rotation de 2 radians ne soit effectuée. Dans le second cas, la translation est d'abord effectuée puis la rotation et le changement d'échelle, tous deux autour du point (20, 20). La deuxième occurence de l'option *–o* est inutile puisqu'elle n'est suivie d'aucune autre option.

Le nom de défaut du golygone résultant d'une opération unaire est le même que celui du golygone d'entrée à moins que l'opération de complémentarisation ne soit demandée. Dans ce cas, le nom sera précédé d'un tilde (~). Cependant, il est possible d'imposer le nom du golygone résultant en utilisant l'option *–N* suivie du nom voulu.

La commande *binop* permet d'effectuer les opérations binaires d'union et d'intersection. Les golygones d'entrée peuvent être normalisés ou non. S'ils ne le sont pas, ou si l'utilisateur n'est pas certain qu'ils le sont, l'option *–n* sert à réaliser cette normalisation. Au contraire des deux commandes précédentes, le paramètre obligatoire ne peut être une combinaison d'opérations. Il sera soit égal à **u**, dans le cas où c'est l'union qui est désirée, soit égal à **i**, dans le cas de l'intersection. La commande qui a servi a obtenir le golygone de la figure 1.4 est:

```
unop c lacs | binop i suisse - | binop i - cath
```

où le fichier *cath* contient le golygone décrivant les régions à majorité catholique et a été obtenu par complémentarisation du fichier *prot*.

A moins de spécification contraire (option *–N*), le nom du golygone résultant est formé en juxtaposant:

le nom du premier golygone d'entrée,

le caractère spécial indiquant l'opération effectuée,

le nom du second golygone d'entrée.

Le caractère spécial indiquant l'union est la barre verticale (|) et celui indiquant l'intersection est la perluète (&).

Toutes les commandes que nous venons de décrire s'attendent à ce que chaque fichier d'entrée contienne la description externe d'**un** golygone, telle que décrite à la section III.1.1. Les utilitaires *gcat* et *gcut* servent, respectivement, à concaténer plusieurs golygones en un seul et à couper un golygone contenant plusieurs composantes en deux golygones ou plus. Le nom du golygone obtenu après un appel à *gcat* est formé par juxtaposition des noms des golygones d'entrée, chacun de ces noms étant séparé du suivant par la barre horizontale (_). Les golygones obtenus après un appel à *gcut* portent le nom du golygone de départ avec un numéro comme extension. Cette commande a des options qui permettent de choisir le nombre de polygones qui composeront les nouveaux golygones. De plus, elle offre la possibilité d'extraire, du golygone d'origine, un sous-ensemble spécifique de composantes formant ces nouveaux golygones. Ainsi, la commande

```
gcut -#5 -10 lacs
```

sert à créer, à partir du golygone contenu dans le fichier *lacs*, et en commençant par la dixième composante, des golygones contenant chacun cinq composantes, alors que

```
gcut -4 +6 lacs
```

sert à isoler les quatrième, cinquième et sixième composantes du golygone contenant les lacs, pour donner les trois golygones suivants:

```
LACS.1
%
695.160000 246.240000
695.160000 245.460000
695.950000 244.270000
696.340000 243.490000
697.130000 241.910000
697.520000 242.700000
697.130000 245.060000
695.950000 245.460000
$
LACS.2
%
689.250000 220.630000
690.430000 219.850000
691.610000 219.850000
692.800000 219.450000
693.580000 218.660000
694.370000 219.450000
693.980000 220.240000
692.800000 221.420000
690.430000 221.820000
$
LACS.3
%
516.680000 185.570000
517.470000 187.540000
518.650000 189.110000
518.650000 189.900000
516.680000 189.110000
515.890000 186.750000
515.100000 184.780000
515.100000 184.390000
$
```

Enfin, le format externe que nous avons imposé ne doit pas représenter une contrainte pour un utilisateur qui voudrait se servir de notre outil en conjonction avec un autre logiciel. En fait, il est facile de faire des filtres permettant de passer d'un format externe à un autre. C'est dans cet esprit que nous avons développé deux

utilitaires permettant de passer du format golygone au format des fichiers *annotations* du logiciel *EMME/2* (EMME/2 User's Manual, 1988). La raison de ce choix est double. Premièrement, ces fichiers sont faciles à interpréter graphiquement. En effet, ils contiennent une séquence de commandes simples ressemblant beaucoup à celles utilisées pour le contrôle d'une table traçante. Deuxièmement, *EMME/2* étant un logiciel interactif-graphique, nous pouvions, par l'intermédiaire des annotations, utiliser ses capacités d'affichage.

Ces utilitaires s'appellent **ggraf** (du format golygone au format commandes graphiques) et **gmake** (du format commandes graphiques au format golygone). L'affichage d'un golygone nécessite deux types de commandes graphiques: une pour demander le positionnement à une certaine adresse, et une pour commander le tracé d'un vecteur de la position courante jusqu'à une nouvelle adresse. Ainsi, la commande *m* (''move'') correspond au début d'une composante polygonale et doit être suivie des coordonnées *x* et *y* du premier sommet de ce polygone, alors que la commande *d* (''draw'') est suivie des cordonnées indiquant la fin d'un segment de droite (sommet suivant). Par exemple, en utilisant le filtre *ggraf* sur le fichier décrivant le format externe du golygone de la figure 2.1 (le contenu de ce fichier est décrit à la section III.1.1), nous obtenons la sortie suivante:

```
m   20.00   20.00
d   60.00   20.00
d   60.00   60.00
d   20.00   60.00
d   20.00   20.00
m   24.00   40.00
d   40.00   24.00
d   56.00   40.00
d   40.00   56.00
d   24.00   40.00
```

Il faut remarquer que, contrairement au format d'entrée, chaque composante est fermée explicitement et que, par conséquent, les coordonnées du premier sommet de chaque composante apparaissent deux fois, la première fois suivant la commande *m* et correspondant à la queue du premier segment, la deuxième fois suivant la dernière

commande *d* de la composante et correspondant à la tête du dernier segment.

Quoique ces deux commandes soient suffisantes pour passer d'un format à l'autre, d'autres commandes graphiques intéressantes, telles le choix de la couleur, du patron de ligne ainsi que du type d'unités pour les coordonnées, peuvent être ajoutées par l'intermédiaire des options offertes par l'utilitaire *ggraf*. De plus, l'option **–f** (*filling*) permet, lors de la création du fichier contenant les commandes graphiques, de générer aussi des segments de hachurage servant à mettre en évidence les régions d'appartenance.

Quant à la commande *gmake*, elle trouve toutes les composantes contenues dans un fichier sous format annotation et donne en retour un golygone sous format externe.

Comme dernier exemple, les figures 3.5 et 3.6 montrent deux annotations (annota et annotb) crées par l'intermédiaire du logiciel *EMME/2*. La figure 3.7 contient le résultat de l'union entre les deux golygones correspondant à ces annotations et a été obtenue par la suite de commandes suivante:

```
gmake annota > gol_tmp
gmake annotb | binop – n u gol_tmp – | ggraf – c 3 – f 4 8  > annotc
```

Enfin, la figure 3.8 contient le résultat de l'intersection entre ces deux mêmes golygones.

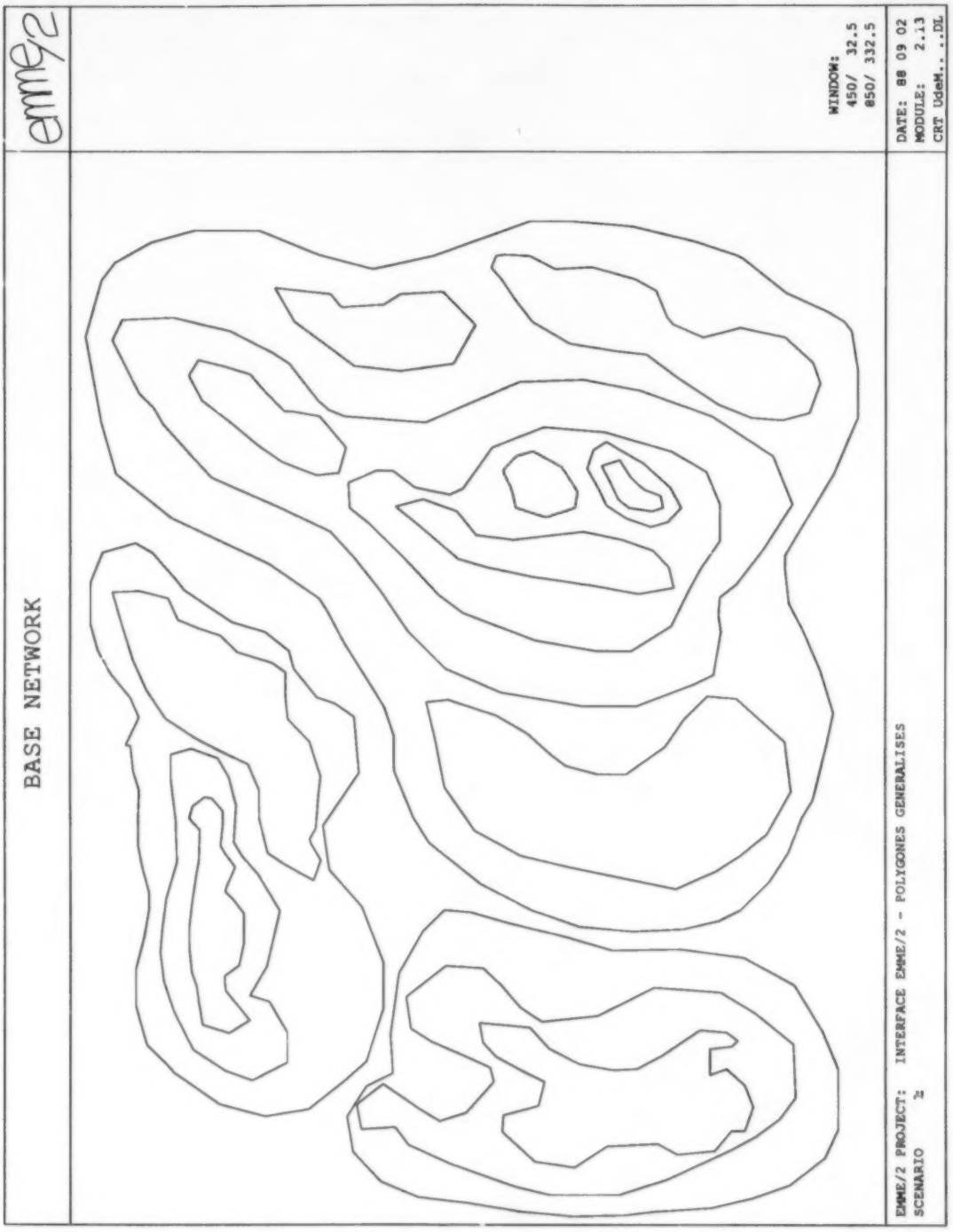


Figure 3.5 Annotation créée avec le logiciel EMME/2 - Golygone A

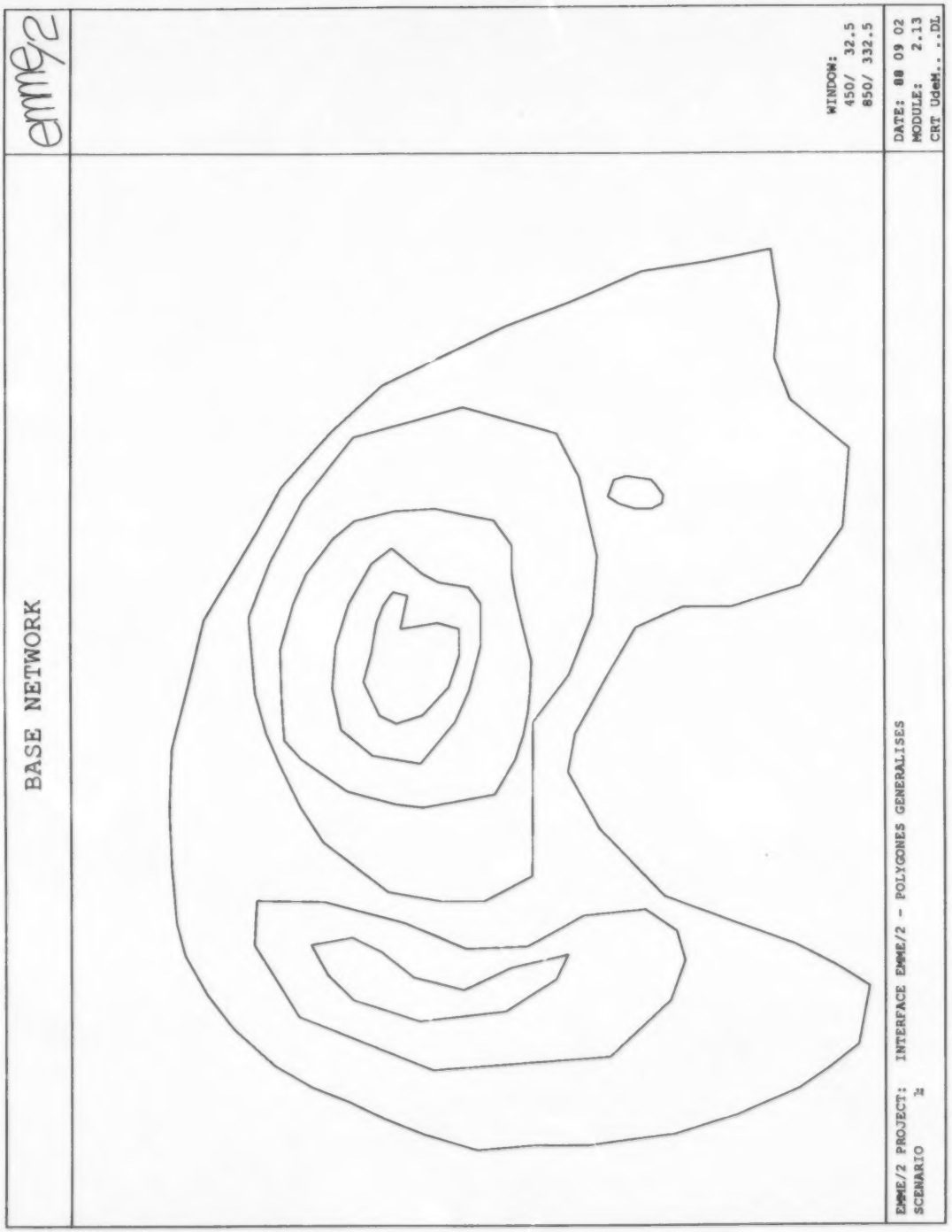


Figure 3.6 Annotation créée avec le logiciel EMME/2 - Golygone B

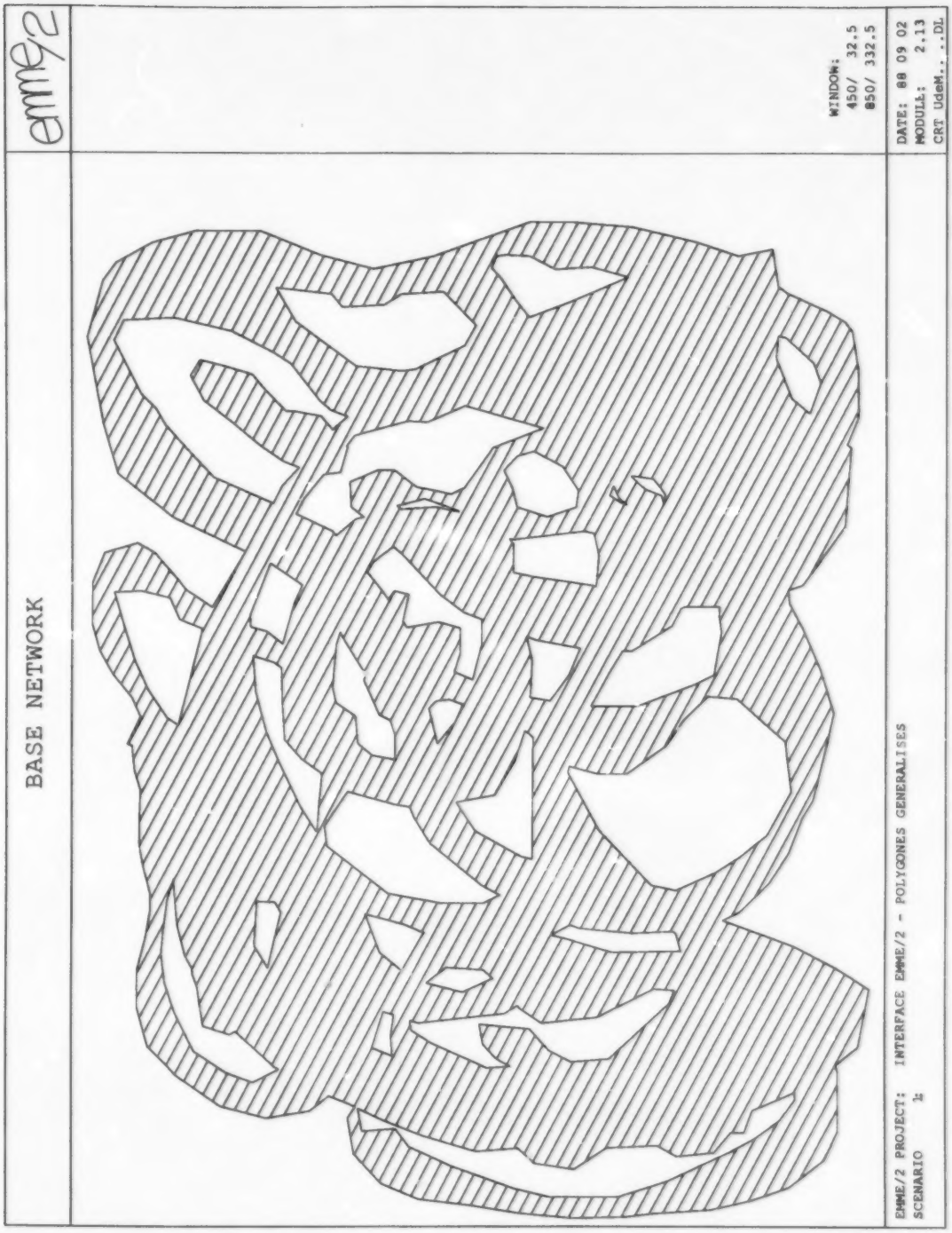


Figure 3.7 Résultat de l'union entre les golygones A et B

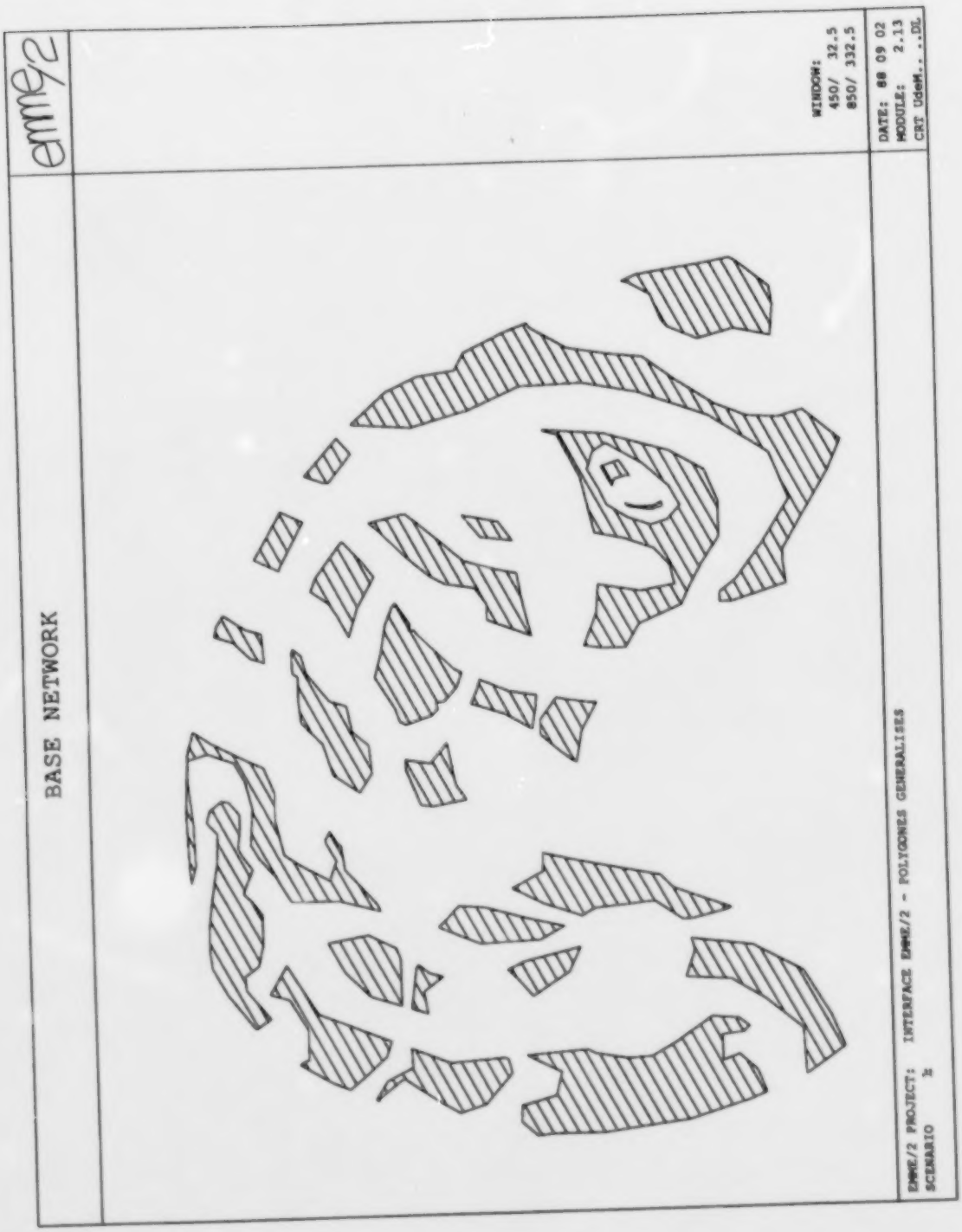


Figure 3.8 Résultat de l'intersection entre les golygones A et B

CONCLUSION

Ce mémoire avait deux objectifs. Le premier était d'examiner le problème complexe qui consiste à effectuer des opérations sur ce que nous avons appelé des polygones généralisés. Ces derniers correspondent à des ensembles de polygones qui servent à partitionner l'espace bidimensionnel en régions, possiblement composées, d'appartenance et de non-appartenance. Le second objectif était d'appliquer les résultats théoriques obtenus en transposant les algorithmes proposés en procédures informatiques.

Bien que nous nous soyons intéressés à trois grandes classes d'opérations sur les polygones généralisés, le traitement des opérations logiques représente la contribution principale de ce mémoire. En effet, tous les ouvrages que nous avons consultés ne traitent que des opérations logiques sur deux polygones simples, et il semble sous-entendu que la généralisation de ces algorithmes à n'importe quel ensemble de polygones ne pose pas de problème. Or, nous avons démontré que cette généralisation n'est ni simple, ni directe. En particulier, puisque les opérations logiques sur deux polygones peuvent fréquemment aboutir à un résultat qui n'est pas lui-même un polygone, nous nous sommes trouvés devant l'impossibilité d'enchaîner facilement plusieurs opérations logiques. Notre insatisfaction devant cet état de fait a constitué une des principales motivations de ce projet et c'est en cherchant à généraliser la classe des objets considérés que nous avons défini le concept de polygones généralisés.

Malgré l'apparence théorique de ce travail, nos motivations étaient éminemment pratiques et les résultats obtenus représentent un apport significatif dans plusieurs domaines d'application. Nous pensons, entre autres, qu'ils pourraient servir dans des études de planification de l'utilisation du territoire et pour l'analyse de données socio-économiques associées à des régions définies de façon géographique.

Nous avons donc intégré les résultats de notre étude dans un outil informatique. Quoique le développement de cet outil se soit fait dans un cadre académique, nous avons mis beaucoup d'efforts dans sa conception afin qu'il puisse facilement évoluer.

En effet, notre but était d'offrir un outil qui non seulement serait immédiatement utilisable, mais qui préparerait aussi le terrain pour des améliorations futures, ainsi que pour le traitement de problèmes connexes.

Pour compléter l'outil et pour agrandir son champ d'utilité, nous envisageons poursuivre les recherches et le développement, principalement selon les avenues suivantes: l'optimisation de certaines procédures; l'étude des cas de dégénérescence non encore traités avec succès; l'ajout de certaines opérations dont la plus importante nous paraît être le nettoyage des données d'entrée afin d'enlever les points inutiles et les composantes non significatives; l'intégration d'un système interactif-graphique à l'outil, afin de faciliter l'affichage et la manipulation des polygones généralisés et des informations qu'ils représentent.

Toutefois, ces développements dépassent largement les objectifs, forcément limités, que nous nous étions fixés dans le cadre de ce mémoire. Nous croyons cependant que notre contribution aura permis de jeter une lumière nouvelle sur le traitement des informations associées à des partitions de l'espace à deux dimensions et qu'elle pourra servir de base pour de nouveaux développements dans ce domaine qui semble riche de promesses.

REFERENCES

Bentley J.L., Ottmann T.A. (1979), *Algorithms for reporting and counting geometric intersections*, **IEEE Transactions on Computers** 28, 643-647.

Bowyer A., Woodwark J.R. (1983), *A Programmer's Geometry*, Butterworths, London.

EMME/2: User's Manual (1988), Centre de recherche sur les transports, Université de Montréal.

Foley J.D., Van Dam A. (1984), *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, USA.

Harrington S. (1983), *Computer Graphics*, McGraw-Hill, USA.

Hohenbalken B.V. (1985), *An Algorithm for Set Operations on Nonconvex Polygons*, Département de science économique et centre de recherche et développement en économique, Université de Montréal, Cahier #8556.

Kernighan B.W., Ritchie D.M. (1978), *The C Programming Language*, Laboratoires Bell, Prentice-Hall, Englewood Cliffs, New Jersey.

Knott G.D. (1987), *Computing Polygon Fill-Lines*, **Computers & Graphics** Vol. 11, No. 1, Pergamon Press, 21-25.

Knuth D.E. (1973), *The Art of Computer Programming*, Vol. 1, 2ième édition, Addison-Wesley.

Magnenat-Thalmann N. et Thalmann D. (1983), *Informatique graphique: concepts et techniques avec le langage MIRA*, Gaétan Morin éditeur, Chicoutimi, Québec, Canada.

O'Rourke J., Chien C., Olson T., Naddor D. (1982), *A new linear algorithm for intersecting convex polygons*, **Computer Graphics and Image Processing** 19, 384-391.

Pavlidis T. (1982), *Algorithms for Graphics and Image Processing*, Computer Science Press, USA.

Shamos M.I., Hoey D. (1976), *Geometric intersection problems* dans *Seventeenth Annual IEEE Symposium on Foundations of Computer Science*, 208-215.

Shamos M.I., Preparata F.P. (1985), *Computational Geometry - An Introduction*, David Gries (ed.), Springer-Verlag, New-York.

SPANS (1988), *An Introduction to Geographic Information Systems*, Tydac Technologies Inc., présenté au *EMME/2 Transportation/Planning Workshop*, Ottawa - Hull, 8 - 10 juin 1988.

Stoll R.R. (1963), *Set Theory and Logic*, W.H. Freeman and Company, San Francisco.

Thomas R. (1983), *What Is a Software Tool?*, **Byte** Vol. 8, No. 8, McGraw-Hill Publication, 222-238.

Thomas R., Rogers L.R. et Yates J.L. (1986), *Advanced Programmer's Guide to UNIX System V*, Osborne, McGraw-Hill, Berkeley, California.

Toussaint G.T. éditeur (1985), *Computational Geometry*, North-Holland, Amsterdam, The Netherlands. David Gries (ed.), Springer-Verlag, New-York,

Toussaint G.T. (1987), *Communication privée*, School of Computer Science, Université McGill, Montréal.

UNIX Programmer's Manual (1983), Vol. 1, Bell Laboratories, Murray Hill, New Jersey.

APPENDICE A

MANUEL D'UTILISATION DES COMMANDES

NOM

binop – effectue une opération logique sur deux golygones

SYNOPSIS

binop [–nv[N name]] op file1 file2

DESCRIPTION

La commande *binop* permet d'effectuer les opérations binaires d'union et d'intersection. Elle doit recevoir en entrée deux golygones (valides ou non) et donne un golygone en sortie. Deux options sont disponibles:

–n (*normalisation*); teste si les golygones d'entrée sont simples et, le cas échéant, s'occupe, non seulement de vérifier l'orientation de chacun des polygones, mais aussi de la modifier s'il y a lieu.

–v (*verbose*); écrit, sur le fichier standard d'erreur (**stderr**), le nombre de composantes dans le golygone résultant.

–N (*name*); spécifie le nom du golygone résultant. Elle doit être suivie de la valeur alphanumérique de remplacement. Par défaut, *binop* nomme le golygone résultant en fonction des noms des golygones de départ et de l'opération demandée.

Le premier argument doit indiquer l'opération désirée: **u** pour l'union ou **i** pour l'intersection. On ne peut pas demander les deux opérations sur la même ligne de commande.

S'ils sont présents, les deux derniers arguments doivent spécifier les deux fichiers d'entrée contenant chacun un golygone. Lorsqu'un des deux fichiers est remplacé par un –, ou lorsqu'aucun fichier n'est précisé, le fichier d'entrée standard (**stdin**) est utilisé.

La sortie est envoyée sur le fichier de sortie standard (**stdout**). Les formats d'entrée et de sortie sont les mêmes.

Si les deux golygones d'entrée sont normalisés, le golygone résultant l'est aussi.

EXEMPLE

Pour faire l'union de trois golygones situés respectivement dans les fichiers *gol1* , *gol2* et *gol3* :

binop u gol1 gol2 | binop u - gol3

VOIR AUSSI

g_binop(3).

NOM

gcat – concatène plusieurs golygones

SYNOPSIS

gcat file1 ... filen

DESCRIPTION

Gcat lit en séquence les golygones contenus dans les fichiers *file*[1...n] et les concatène en un seul. Le résultat est écrit sur le fichier de sortie standard.

Il est important de noter que le nouveau golygone est formé de toutes les composantes des golygones d'entrée sans qu'aucune validation ne soit faite. Par conséquent, le golygone obtenu peut ne pas être simple.

VOIR AUSSI

gcut(1).

NOM

gcut – coupe un golygone en plusieurs

SYNOPSIS

gcut [–#N] [–N1] [+N2] [file]

DESCRIPTION

Gcut permet d'isoler, d'un golygone de départ, un ou plusieurs blocs de polygones. Par défaut, un golygone de *m* composantes est divisé en *m* golygones de 1 composante. Cependant il est possible de spécifier, grâce aux options **–N1** et **+N2**, quelles sont les composantes à considérer dans le golygone d'entrée. N1 et N2 sont alors des entiers représentant respectivement l'indice de la première et de la dernière de ces composantes. La valeur de défaut pour N1 est 1 alors que celle pour N2 est égale au nombre de polygones constituant le golygone de départ.

Lorsqu'on utilise l'option **–#N**, chacun des nouveaux golygones sera formé de N composantes sauf, éventuellement, le dernier qui pourra en contenir moins.

REMARQUE

Pour le moment la sortie est écrite sur le fichier de sortie standard, mais dans la prochaine version il sera possible d'avoir un fichier par golygone obtenu.

VOIR AUSSI

gcat(1).

NOM

ggraf – traduit un golygone en une suite de commandes graphiques simples

SYNOPSIS

ggraf [– [u][f dx dy][l pattern][c color]] [if1]

DESCRIPTION

Ggraf est un filtre qui sert à transformer le format externe d'un golygone en une suite de commandes **m** (*move*) *x y* et **d** (*draw*) *x y*. Un *m* marque le début d'une composante polygonale alors qu'un *d* correspond à la fin d'un segment de droite. Les différentes options disponibles sont:

–u (*unit*); permet que les coordonnées soient interprétées dans le système de coordonnées de l'utilisateur. Par défaut, ce sont des coordonnées écran.

–f (*filling*); permet de générer aussi des segments de hachurage pour remplir les régions d'inclusion du golygone. Cette option doit être suivie de deux arguments décrivant le vecteur qui spécifie la distance entre deux lignes de hachurage.

–l (*line*); permet de spécifier le patron à utiliser pour tracer les lignes. Cette option doit être suivie d'un paramètre entier correspondant à l'indice du patron désiré. Le patron de défaut est la ligne pleine.

–c (*color*); permet de spécifier la couleur à utiliser pour tracer le golygone. Cette option doit être suivie d'un paramètre entier correspondant à l'indice de la couleur désirée. Les couleurs sont dépendantes du terminal utilisé.

Si aucun fichier d'entrée n'est spécifié, le fichier d'entrée standard est utilisé.

REMARQUE

Les options offertes sont fortement liées au système graphique sous-jacent, dans notre cas le système EMME/2.

VOIR AUSSI

gmake(1).

NOM

gmake – traduit une suite de commandes graphiques en golygone externe

SYNOPSIS

gmake [–N name] fichier

DESCRIPTION

Gmake est un filtre qui transforme un ensemble de commandes *m x y* et *d x y* en un golygone sous format externe. La seule option disponible est –N (*name*) et elle sert à changer le nom du golygone résultant. Elle doit être suivie de la valeur alphanumérique de remplacement.

ATTENTION

Gmake ne peut produire que des composantes fermées. Ainsi une polyligne sera transformée en polygone.

VOIR AUSSI

ggraf(1).

NOM

scalop – effectue une opération scalaire sur des golygones

SYNOPSIS

scalop [–v] acnps[i xi yi] if1 [if2...ifn]

DESCRIPTION

La commande *scalop* permet d'effectuer des opérations scalaires sur un ou plusieurs golygones d'entrée. Le premier argument est formé par une combinaison de lettres spécifiant les opérations désirées:

a (*area*); utilisé pour obtenir l'aire du golygone.

c (*component*); donne le nombre de composantes dans le golygone.

i (*inclusion*); sert à tester l'inclusion naïve d'un point dans un golygone. Si elle est utilisée, cette opération doit toujours être la dernière spécifiée et suivie de deux arguments donnant les coordonnées du point. Si le golygone est simple et normalisé, la valeur d'inclusion sera 0 si le point est exclus, 1 s'il est inclus dans un golygone fini et –1 s'il est inclus dans un golygone infini. Sinon, la valeur retournée est un entier reflétant le degré d'enroulement du golygone autour du point.

n (*node*); donne le nombre de sommets ou segments dans le golygone.

p (*perimeter*); utilisé pour obtenir le périmètre du golygone.

s (*simplicity*); sert à tester la simplicité d'un golygone en vérifiant qu'aucun segment n'en intersecte un autre. Une valeur 1 indique la simplicité et 0 la non-simplicité.

Les valeurs calculées sont écrites sur le fichier de sortie standard. Par défaut, *scalop* génère une ligne par golygone d'entrée donnant en première colonne le nom du golygone, suivi des valeurs correspondant aux opérations désirées. Indépendamment de l'ordre dans lequel ces opérations ont été demandées sur la

ligne de commande, les résultats appropriés apparaissent toujours dans l'ordre suivant: nombre de composantes, nombre de segments, simplicité, aire, périmètre et valeur d'inclusion. Lorsque l'option **-v** (*verbose*) est utilisée, le format de sortie est plus explicite.

VOIR AUSSI

g_area(3), g_incl(3), g_perim(3), g_simple(3).

NOM

unop – modifie un golygone

SYNOPSIS

unop [–v] [–o xo yo] [–x hx] [–y hy] [–r alpha] [–t tx ty] [–s sx sy] [–m a b c d e f] [–N name] lnc [ifile]

DESCRIPTION

La commande *unop* permet d'effectuer des opérations unaires c'est-à-dire qu'elle accepte un golygone en entrée et donne un autre golygone en sortie. Trois types d'opérations unaires sont disponibles: la complémentarisation, la normalisation et la transformation linéaire

A l'exception de l'option **–v** (*verbose*), les options servent à définir la transformation linéaire:

–o (*origin*); pour modifier le point d'origine courant. L'option doit être suivie des coordonnées x et y. La valeur de défaut est (0,0).

–x pour désaxer (*shear*) le golygone dans le sens horizontal. Cette option doit être suivie d'une valeur réelle.

–y pour désaxer (*shear*) le golygone dans le sens vertical. L'option doit être suivie d'une valeur réelle.

–t (*translate*); pour demander une translation de chacune des composantes. Cette option doit être suivie de deux valeurs réelles précisant, pour la première, le déplacement en x, et pour la seconde, le déplacement en y.

–s (*scale*); pour changer l'échelle de toutes les composantes. L'option doit être suivie de deux valeurs réelles correspondant aux facteurs multiplicatifs en x et y.

–r (*rotate*); pour faire la rotation des composantes du golygone. Cette option doit être suivie d'un paramètre spécifiant l'angle de rotation (en radians).

–m (*matrix*); permet de donner directement les valeurs de la matrice de transformation. Elle doit être suivie de six valeurs réelles constituant la matrice de transformation. Lorsque l'option **–v** est utilisée, les valeurs de la matrice sont affichées sur le fichier standard d'erreur.

–N (*name*); pour changer le nom du golygone. Elle doit être suivie de la valeur alphanumérique de remplacement.

Les transformations linéaires peuvent être combinées et sont appliquées selon l'ordre d'entrée des options. Lorsque le point d'origine courant est modifié, il s'applique à toutes les transformations qui suivent jusqu'à ce qu'il soit de nouveau modifié.

L'argument **l** (*linear transformation*), doit toujours suivre une suite d'options de transformation. Lorsqu'il n'est combiné à aucune option, il permet l'opération d'identité.

L'argument **n** (*normalization*) est utilisé lorsque l'on veut orienter correctement chacun des polygones constituants en fonction de son niveau d'inclusion dans l'arborescence décrivant le golygone.

L'argument **c** (*complement*) indique que l'on veut changer l'orientation de chacune des composantes du golygone d'entrée.

Des opérations combinées sont toujours appliquées dans l'ordre **l**, **n**, **c**.

Si le dernier argument de la commande est un fichier, celui-ci contient les données d'entrée, sinon le fichier d'entrée standard est utilisé.

EXEMPLES

Pour normaliser le golygone d'entrée et doubler la taille de chacune de ses composantes:

```
unop –s 2 2 ln gol1
```

Pour bâtir une transformation complexe et avoir ensuite les valeurs de la matrice utilisée:

```
unop –v –t 10 10 –o 10 10 –s 0.5 0.5 –o 0 0 –r 0.25  1 gol2
```

VOIR AUSSI

g_cmpl(3), g_linear(3), g_norm(3).

APPENDICE B

MANUEL D'UTILISATION DES PROCEDURES

NOM

cros_seg – vérifie s'il y a intersection entre deux segments

SYNOPSIS

#include "golygon.h"

CROSS *cros_seg (sa, sb)
SEGMENT *sa, *sb;

DESCRIPTION

Cros_seg vérifie si les deux segments reçus comme paramètres intersectent. Si c'est le cas, cette fonction alloue l'espace mémoire pour une structure de type *CROSS* et remplit les champs *xi*, *yi* (coordonnées du point d'intersection), *flag* et les deux valeurs calculées de *lambda*. Elle retourne ensuite un pointeur vers cette structure.

Si les deux segments sont parallèles ou s'ils n'intersectent pas, elle retourne un pointeur nul. S'il n'y a plus d'espace mémoire, elle émet un message d'avertissement et retourne la valeur −1.

VOIR AUSSI

binop(1), scalop(1).

NOM

g_area – calcule la surface d'un golygone

SYNOPSIS

#include "golygon.h"

double g_area (gol)
GOLYGON *gol;

DESCRIPTION

G_area retourne une valeur réelle correspondant à la somme des surfaces de toutes les composantes du golygone reçu en entrée. Si la valeur retournée est positive, cela signifie que la somme des surfaces des polygones appartenant au golygone et tournant en sens horaire est plus petite que celle des autres polygones. Lorsque le golygone est dégénéré, elle retourne la valeur 0.

Ni normalisation, ni simplicité du golygone n'est requise.

VOIR AUSSI

scalop(1), p_area(3).

NOM

g_binop – réalise une opération logique entre 2 golygones

SYNOPSIS

```
#include "golygon.h"

int g_binop(ofile, gol, dir)
FILE *ofile;
GOLYGON *gol[2];
int dir;
```

DESCRIPTION

G_binop génère le nom du polygone résultant puis cherche tous les points d'intersection entre les deux golygones. Elle monte ensuite l'arborescence d'inclusion, si ce n'est déjà fait et s'il existe des composantes f-disjointes. Enfin, elle exécute l'opération binaire (logique) spécifiée par *dir*. Pour l'union, la valeur de *dir* doit être égale à 1, alors que pour l'intersection, elle sera égale à 0. Le résultat est généré dans le fichier spécifié par *ofile*.

G_binop retourne un entier donnant le nombre de polygones composant le golygone résultant.

VOIR AUSSI

binop(1), cros_seg(3), g_cross(3), g_print(3), g_tree(3).

NOM

g_cmpl – complémente un golygone

SYNOPSIS

#include "golygon.h"

int g_cmpl(gol)
GOLYGON *gol;

DESCRIPTION

G_cmpl complémente un golygone, c'est-à-dire fait tourner en sens inverse chacune de ses composantes.

Ni normalisation, ni simplicité du golygone n'est requise.

DIAGNOSTICS

Si tout se passe normalement, *g_cmpl* retourne la valeur 0, sinon retourne la valeur 1.

VOIR AUSSI

unop(1), p_cmpl(3).

NOM

g_cross – cherche tous les points d'intersection entre 2 golygones

SYNOPSIS

#include "golygon.h"

int g_cross(gol)
GOLYGON *gol[2];

DESCRIPTION

G_cross cherche tous les points d'intersection entre les segments de deux golygones. Pour chaque intersection trouvée, une structure de type *CROSS* est allouée. Les indices des deux segments impliqués sont placés respectivement dans *cross->iseg*[0], pour le segment appartenant à *gol*[0], et dans *cross->iseg*[1] pour le segment appartenant à *gol*[1]. Les intersections sur un même segment sont triées par ordre croissant de λ.

Si tout se passe bien, cette fonction retourne un entier positif ou nul correspondant au nombre de composantes f-disjointes, c'est-à-dire celles dont aucun segment n'intersecte avec les segments de l'autre golygone. Sinon, elle retourne la valeur −1.

Ni normalisation, ni simplicité des golygones n'est requise.

VOIR AUSSI

binop(1), cros_seg(3), g_binop(3).

NOM

g_incl – teste l'inclusion d'un point dans un golygone

SYNOPSIS

```
#include "golygon.h"

int g_incl(x, y, gol)
GOLYGON *gol;
double x, y;
```

DESCRIPTION

G_incl teste l'inclusion, dans un golygone, d'un point spécifié par ses coordonnées x et y. La signification de la valeur retournée est dépendante du type du golygone d'entrée. Pour un golygone normalisé fini, *g_incl* retourne la valeur 1, si le point est inclus, 0 autrement. Pour un golygone normalisé infini, elle retourne la valeur 0, si le point est inclus naïvement, −1 autrement. Enfin, lorsque la valeur retournée est supérieure à 1, ou inférieure à −1, cela signifie que le golygone d'entrée n'est pas simple ou n'est pas normalisé.

VOIR AUSSI

scalop(1), p_incl(3).

NOM

g_linear – opère une transformation linéaire sur un golygone

SYNOPSIS

```
#include "golygon.h"

int g_linear(tmat, gol)
TRANSMAT *tmat;
GOLYGON *gol;
```

DESCRIPTION

G_linear sert à appliquer *in situ* une même transformation linéaire à tous les sommets x, y d'un golygone. Les valeurs utilisées pour la transformation sont contenues dans la structure pointée par *tmat*.

```
typedef struct transmat {
   double  a, c, e;
   double  b, d, f;
} TRANSMAT;
```

Les équations utilisées pour l'obtention des nouvelles valeurs x', y' sont:

$$x' = ax + cy + e$$

$$y' = bx + dy + f$$

La normalisation et la simplicité ne sont pas requises en entrée mais ne sont pas non plus assurées en sortie.

DIAGNOSTICS

Si tout se passe normalement, *g_linear* retourne la valeur 0, sinon retourne la valeur 1.

VOIR AUSSI

unop(1).

NOM

g_norm – normalise un golygone

SYNOPSIS

#include "golygon.h"

int g_norm(gol)
GOLYGON *gol;

DESCRIPTION

G_norm vérifie et corrige, au besoin, l'orientation de chacune des composantes du golygone d'entrée en suivant la règle suivante: tout polygone inclus dans un nombre impair de composantes doit tourner en sens horaire, et inversement. Pour savoir le niveau d'inclusion de chacune des composantes, *g_norm* monte l'arborescence d'inclusion du golygone d'entrée, si ce n'est pas déjà fait.

Le golygone d'entrée doit être simple.

DIAGNOSTICS

G_norm retourne une valeur entière positive ou nulle correspondant au nombre de composantes dont l'orientation a été changée. Si la variable globale *verbose* est utilisée, un message est émis lorsqu'une composante n'était pas correctement orientée.

VOIR AUSSI

binop(1), unop(1), g_tree(3), p_cmpl(3).

NOM

g_perim – calcule le périmètre d'un golygone

SYNOPSIS

#include "golygon.h"

double g_perim (gol)
GOLYGON *gol;

DESCRIPTION

G_perim retourne une valeur réelle positive correspondant au périmètre du golygone reçu en entrée. Pour les cas dégénéres $\mathbb{R}^2$ ou $\emptyset$, la fonction retourne la valeur 0.

Ni normalisation, ni simplicité du golygone d'entrée n'est requise.

VOIR AUSSI

scalop(1), p_perim(3).

NOM

g_print – donne en sortie les coordonnées d'un golygone

SYNOPSIS

#include "golygon.h"

int g_print(ofile, gol)
FILE *ofile;
GOLYGON *gol;

DESCRIPTION

G_print écrit dans le fichier de sortie spécifié par *ofile* les coordonnées des sommets de chacune des composantes constituant le golygone d'entrée. Le résultat est généré dans un format compatible avec celui utilisé par *g_read* c'est-à-dire que le début de chaque composante est annoncé par une ligne commençant par le caractère ''%'' et que le dernier caractère écrit est celui de fin de golygone (''$'').

Simplicité et normalisation du golygone ne sont pas requises.

DIAGNOSTICS

Si tout se passe normalement, *g_print* retourne la valeur 0, sinon retourne la valeur 1.

VOIR AUSSI

unop(1), g_binop(3), g_read(3), p_print(3).

NOM

g_read – lit un golygone

SYNOPSIS

```
#include "golygon.h"

int g_read(ifile, gol)
FILE ifile;
GOLYGON *gol;
```

DESCRIPTION

G_read sert à lire sur le fichier d'entrée spécifié par *ifile*, les données décrivant un golygone. Le format du fichier d'entrée doit suivre les règles suivantes: la première ligne contient toujours le nom du golygone; une composante polygonale est décrite comme une suite de lignes, chacune contenant les coordonnées *x* et *y* d'un sommet; entre chaque composante polygonale, il y a une ligne contenant "%" comme premier caractère; le caractère "$" en début de ligne marque la fin du golygone. Toute information supplémentaire sur une ligne reconnue comme valide est ignorée.

C'est la fonction *g_read* qui alloue l'espace nécessaire pour tenir en mémoire toutes les informations concernant un golygone. Les coordonnées des points lus sont placées dans le tableau des segments, les valeurs de déplacement entre deux points consécutifs sont calculées, et tous les champs ne contenant aucune valeur calculée sont initialisés.

DIAGNOSTICS

Lorsque tout fonctionne bien, *g_read* retourne la valeur 0. Dès qu'une ligne d'entrée ne correspond pas au format attendu, la fonction arrête la lecture et retourne la valeur 1. S'il manque d'espace mémoire, elle retourne la valeur −1.

VOIR AUSSI

binop(1), scalop(1), unop(1), g_print(3).

NOM

g_simple – vérifie la simplicité d'un golygone

SYNOPSIS

#include "golygon.h"

int g_simple(gol)
GOLYGON *gol;

DESCRIPTION

G_simple vérifie si un golygone est simple. Si aucun des segments composant le golygone n'intersecte avec aucun autre, la fonction retourne la valeur 1. Sinon, la valeur retournée est 0. Deux segments consécutifs dans une même composante sont considérés comme n'intersectant pas.

La normalisation du golygone d'entrée n'est pas requise.

VOIR AUSSI

binop(1), scalop(1), cros_seg(3).

NOM

g_tree – monte l'arborescence d'inclusion des composantes d'un golygone

SYNOPSIS

```
#include "golygon.h"

COMP *g_tree(gol)
GOLYGON *gol;
```

DESCRIPTION

G_tree établit la relation de descendance entre chacune des composantes du golygone d'entrée. Pour chaque composante, elle trouve et emmagasine les valeurs *comp->area* (surface du polygone), *comp->nbcomp* (nombre de polygones englobants), *comp->left* (pointeur vers le premier polygone "fils") et *comp->right* (pointeur vers le premier polygone "frère").

VOIR AUSSI

binop(1), unop(1), g_cmpl(3).

NOM

p_area – calcule la surface d'un polygone

SYNOPSIS

#include "golygon.h"

double p_area (pol)
COMP *pol;

DESCRIPTION

P_area retourne une valeur réelle correspondant à la surface du polygone reçu en entrée. Si la valeur retournée est négative, cela signifie que l'orientation du polygone est en sens horaire.

La simplicité du polygone d'entrée n'est pas requise.

VOIR AUSSI

g_area(3).

NOM

p_cmpl – complémente un polygone

SYNOPSIS

#include "golygon.h"

int p_cmpl(pol)
COMP *pol;

DESCRIPTION

P_cmpl complémente un polygone, c'est-à-dire change l'orientation de chacun de ses segments.

La simplicité du polygone d'entrée n'est pas requise.

DIAGNOSTICS

Si tout se passe normalement, *p_cmpl* retourne la valeur 0, sinon retourne la valeur 1.

VOIR AUSSI

g_cmpl(3).

NOM

p_incl – teste l'inclusion d'un point dans un polygone

SYNOPSIS

```
#include "golygon.h"

int p_incl(x, y, pol)
COMP *pol;
double x, y;
```

DESCRIPTION

P_incl teste l'inclusion naïve, dans un polygone, d'un point spécifié par ses coordonnées x et y. Cette fonction retourne 0 si le point est à l'extérieur. Si le point est à l'intérieur, la valeur retournée par la fonction dépendra du polygone d'entrée. S'il est simple et fini, la fonction retourne la valeur 1. S'il est simple et infini (segments orientés en sens horaire), elle retourne la valeur −1. S'il n'est pas simple, toute valeur entière peut être retournée et cette valeur reflète le degré d'inclusion du point et l'orientation du polygone.

VOIR AUSSI

g_incl(3).

NOM

p_perim – calcule le périmètre d'un polygone

SYNOPSIS

#include "golygon.h"

double p_perim (pol)
COMP *pol;

DESCRIPTION

P_perim retourne une valeur réelle positive ou nulle correspondant au périmètre du polygone reçu en entrée.

La simplicité du polygone d'entrée n'est pas requise.

VOIR AUSSI

g_perim(3).

NOM

p_print – écrit les coordonnées d'un polygone

SYNOPSIS

#include "golygon.h"

int p_print(ofile, comp)
FILE *ofile;
COMP *comp;

DESCRIPTION

P_print écrit sur le fichier de sortie spécifié par *ofile* les coordonnées de tous les sommets du polygone reçu en paramètre.

DIAGNOSTICS

Si tout se passe normalement, *p_print* retourne la valeur 0, sinon retourne la valeur 1.

VOIR AUSSI

g_print(3).